Sistemas Informáticos
Curso 2006-2007

Aventura Gráfica Multijugador

AGM

Jaime Agudo Villanueva
Juan Carlos Castromil Campos
Carlos Pinto Camarero

Dirigido por:
Prof. Jorge J. Gómez Sanz
Dpto. Sistemas informáticos y programación (SIP)

Facultad de Informática
Universidad Complutense de Madrid

Índice

1. Resumen 5
   1.1. Resumen 5
   1.2. Abstract 5
   1.3. Introducción 6
2. Trabajo futuro… 17
   2.1. Introducción 17
   2.2. Preparándose para el entorno… 18
      2.2.1. Lomboz 19
   2.3. Preparando el entorno… 23
      2.3.1. Instalación en Windows XP 23
   2.4. Mantenibilidad de AGM 24
      2.4.1. Log4J 24
      2.4.2. Buenas Prácticas 27
      2.4.3. Documentación de código 33
   2.5. Corrección de Bugs 35
      2.5.1. Consejos para la detección y corrección de bugs 35
      2.5.2. Conversación entre jugador y personaje no jugador 36
      2.5.3. Cierre de las sesiones JMS 38
      2.5.4. Excepción al tiempo de inactividad del servidor 38
      2.5.5. Excepción JMS 40
      2.5.6. Mala programación de ProfesorSOS.java 43
      2.5.7. Actualización de objetos en inventario 44
      2.5.8. Mal cambio de habitación 45
      2.5.9. Mensaje inesperado 49
3. Mejora de la jugabilidad en AGM 54
   3.1. Introducción 54
   3.2. Cambios en la interfaz gráfica 54
      3.2.1. Introducción 54
      3.2.2. Así era la interfaz original 54
      3.2.3. El proceso de reingeniería 57
      3.2.4. Así es la nueva interfaz 61
      3.2.5. Análisis de clases 64
         3.2.5.1. Paquete clienteJugador: Comparativa 64
         3.2.5.2. La clase PanelBienvenida 66
         3.2.5.3. La clase PanelCreacion 66
         3.2.5.4. La clase PanelConexion 67
         3.2.5.5. La clase InterfazGrafica 68
         3.2.5.6. La clase Pantalla 80
         3.2.5.7. La clase ObjetoEnPantalla 81
         3.2.5.8. La clase EscritorTexto 83
         3.2.5.9. La clase HiloComunicacion 84
         3.2.5.10. La clase FragmentoConversacion 85
         3.2.5.11. La clase LineaTiempo 86
         3.2.5.12. La clase ClienteJugador 87
         3.2.5.13. La clase Teclado 96
      3.2.6. Análisis de componentes 96
         3.2.6.1. Paneles de bienvenida, creación y conexión 96

3.2.6.2. Pantalla de juego y consola 102
3.2.6.3. Display 104
3.2.6.4. Caja de acciones 105
3.2.6.5. Inventario 107
3.2.6.6. Chat 109
3.3. Casos de uso 110
3.3.1. Arrancar la aplicación 112
3.3.2. Crear un personaje 113
3.3.3. Conectar un personaje 114
3.3.4. Ejecutar una acción (ir, coger, usar o mirar) 116
3.3.5. Hablar con un personaje 120
3.3.6. Hablar con otros jugadores 123
3.4. Integración de animación 124
3.5. Trabajo futuro 126
4. Framework para la creación de historias en AGM 130
4.1. Resumen introductorio 130
4.2. Diagramas de Clases 131
4.2.1. Paquete vista 131
4.2.1.1. Paquete vista.objetos 132
4.2.1.2. Paquete vista.personajes 133
4.2.1.3. Paquete vista.habitaciones 134
4.2.2. Paquete controlador 135
4.2.3. Paquete modelo 136
4.2.3.1. Paquete modelo.fachada 137
4.2.3.2. Paquete modelo.factoría 138
4.2.3.3. Las clases de tipo Gestión 138
4.2.3.4. Transfer Objeto 139
4.2.3.5. Transfer Personaje 140
4.2.3.6. Transfer Conversación 141
4.2.3.7. Transfer Habitación 142
4.2.3.8. Transfer Puerta 142
4.2.4. Paquete integración 143
4.3. Diagramas de Secuencia 144
4.3.1. Creación de un nuevo Objeto 144
4.3.2. Quitar un personaje 146
4.3.3. Modificar una habitación 147
4.3.4. Siguiente Objeto 149
4.3.5. Captura de coordenadas de un personaje 151
4.4. Conexión con AGM 153
4.4.1. La clase ObjetoEditor 155
4.4.2. La clase PersonajeEditor 162
4.5. Creando una misión paso a paso 170
4.5.1. Guión de la historia 170
4.5.2. Crear una misión nueva 170
4.5.3. Creando las habitaciones 171
4.5.4. Creando los objetos 176
4.5.5. Creando los personajes 180
4.5.6. Importando la misión 186
4.6. Problemas encontrados durante el desarrollo 187
5. Bibliografía 189

6. Palabras clave 189
7. Cesión de derechos 190

1. Resumen

1.1. Resumen

El presente proyecto es una continuación de otros que se realizaron en los años 2003/04 y 2004/05. Se ha dividido en tres grandes bloques:

- Un primer bloque encaminado a que el resto de años en los cuáles se afronte el proyecto el presente documento sea una herramienta esencial para la instalación y seguimiento del mismo. Para ello se proporcionarán instrucciones para la instalación del entorno, uso del entorno, de un sistema de trazas, de un estilo de documentación común y el arreglo de una serie de bugs

- Se ha mejorado la jugabilidad de la AGM replicando la interfaz gráfica de la primera versión. Se ha paliado la ausencia de información sobre la interfaz y su relación con la aplicación, facilitando su futura comprensión y cambio. Se incluye un apartado de mejoras y un estudio de la situación de la animación.

- Framework para la creación de historias en AGM, desarrollo de una herramienta para crear misiones de manera sencilla y rapida. Se explicará como está diseñado el editor mediante diagramas de clase y de secuencia y su conexión con la herramienta original AGM mediante explicación del código fuente fundamental. Y se mostrará un tutorial práctico de cómo crear una misión con el editor paso a paso.

1.2. Abstract

This Project is the continuation of others made during the years 2003/04 and 2004/05. It has been divided into three big blocks:

- The first unit is orientated towards the future years in which the present document would be an essential tool for the installation and the following of its functionality. To do this, guidelines are readily available for the installation of the program, the use of its environment, of a series of touches, papers and the arrangement of a series of bugs.

- The playability of AGM has been improved by imitating the graphic interface of the first version. The lack of information about the interface has been reduced as well as its relationships to the application, and therefore facilitating future comprehension problems and changes. A section with the improvements made and a study of the animation situation are also included.

- Framework for the creation of stories in AGM, the development of a tool able to create missions in a simple, effective and quick way. The manner in which the editor is designed through diagrams of class and sequence and its links with the original AGM tool by the explanation of the main and fundamental code will be explained. Furthermore, a step by step practical tutorial of how to create a mission with the editor will be available.

1.3. Introducción

El primer gran bloque de este proyecto de la asignatura de SSII ha surgido al darse cuenta a la hora de afrontar el proyecto de que no se estaba haciendo correctamente desde un principio.
Tan solo arrancar el proyecto llevó bastante tiempo por distintas dificultades que se encontraron en un principio. Las distintas **dificultades** fueron las siguientes:

1. Dificultad para arrancar la aplicación. En años anteriores la aplicación se desarrollo bajo Linux e incluso se desarrolló mediante conexión a un servidor remoto de GRASIA.
   Esta opción del servidor remoto fue descartado ya que por algún problema técnico que desconocemos se imposibilitó la forma de conexión que había con este servidor.
   Por ello se nos proporcionó en un CD mucho material relacionado con el desarrollo de AGM: Eclipse, Jboss, distintas versiones de AGM, etc…
   El problema es que este material se sacó de una máquina Linux y ninguno de los integrantes del proyecto del curso 2006-2007 disponía de ninguna distribución Linux. Se probaron varias y al final se dio con una en que las versiones Jboss y Eclipse proporcionadas funcionaban sin problemas.
   Entonces se pasó a estudiar a fondo el IDE con Lomboz necesarios para el desarrollo de AGM. Se tuvieron muchos problemas ya que se probaron todas las versiones de AGM que se tenían y ninguna funcionaba correctamente ni tan siquiera para poder llegar a desplegar la aplicación. Se hicieron varios intentos pero debido al desconocimiento del manejo en entornos Linux se decidió instalar todo en Windows.
   Para ello se buscaron las versiones de Eclipse, Lomboz y Jboss similares a las que disponíamos en Linux. Después de mucho tiempo buscando estas versiones, se llegó a 3 que eran compatibles pero el problema se tuvo entonces con el propio código fuente de AGM ya que Lomboz no era capaz de parsear ciertos EJB´s. Después de probar e investigar acerca del fallo, se consiguió arrancar la aplicación pero se había perdido ya mucho tiempo.
2. A la hora de ponerse manos a la obra a investigar el código para ver por dónde enfocar el proyecto en sí, se hizo patente la poca documentación de la cuál dispone el código y esto es algo que debería ser corregido.
3. Además a la hora de ejecutar el juego tan solo en modo 1 jugador se observaron ciertos bugs en la ejecución que deberían ser abordados.

Todo esto dio pie a que el primer bloque de esta documentación sea una herramienta de tremenda utilidad para los siguientes desarrollos que se lleven a cabo en AGM. Se puede dividir en los siguientes **apartados**:

1. Una introducción al desarrollo J2EE con Lomboz de Eclipse. Algo fundamental a la hora de desarrollar AGM es manejar bien el entorno de desarrollo necesario. Se ha dado por hecho en este proyecto que los futuros integrantes de AGM están famliarizados con el desarrollo Java con el IDE Eclipse.
   Además es necesario tener algunas nociones básicas sobre el servidor de aplicaciones Jboss que se usa para desplegar la aplicación web en que consiste

AGM.
Se proporcionará adjunto a este documento una especificación detallada de todo lo concerniente a la versión de Jboss utilizada. En cuanto a Lomboz, se le dedicará un apartado sobre cómo utilizarlo en general:

a. Explicación acerca de la vista correspondiente al plugin de Lomboz en Eclipse.
b. Terminología utilizada en Lomboz a la hora de nombrar distintos componentes J2EE
c. Configuración de los distintos servidores sobre los que es capaz de desplegar componentes J2EE.
d. Pasos para poder llegar a generar EJB, desplegar módulos y arrancar servidores

Como última explicación de este apartado se darán una serie de consejos a la hora de utilizar Lomboz con AGM: algunas especificaciones sobre cómo se ha trabajado durante 2006-2007 para el desarrollo del videojuego.

2. Una serie de instrucciones para la instalación de todo el entorno en una máquina bajo el SO Windows.
   Este apartado es de vital importancia ya que se tiene constancia de que casi la mitad del tiempo de los 2 últimos años de proyecto se perdieron en el arranque de AGM por primera vez.
   De esta forma con tan solo 15 minutos aproximadamente, siguiendo los pasos que se detallan, se puede completar la instalación y se estaría en facultades de arrancar AGM en un entorno Windows.

3. Se incorpora también en esta documentación un tutorial acerca del uso del sistema de trazas de *Apache* "Log4j" para seguir usándolo en AGM debido a las ventajas que proporciona su uso frente al antiguo *"System.out"* de Java.
   Al igual que en el tutorial de Lomboz, se darán una serie de observaciones y explicaciones que detallan el funcionamiento y configuración de Log4j para acabar finalmente con consejos acerca de su uso y configuración dentro de AGM.

4. Además se incluyen una serie de buenas prácticas a seguir a la hora de programar en Java.
   Para cada buena práctica, se explica un breve descripción de la misma, un ejemplo de violación de la buena práctica en el código de AGM y un ejemplo de reparación que se ha llevado a cabo en AGM para que se cumpla..
   El cumplimiento de éstas dan lugar a código más óptimo en la mayoría de los casos y han surgido como análisis del código fuente que ya se había escrito en AGM en años anteriores.

5. Algo esencial en todo proyecto es la documentación del mismo así que se ha seguido una formato *javadoc* de documentación en cada método analizado que da completa información acerca de qué hace el método para así no perder tiempo ni esfuerzo en ir recorriendo el código para entender la lógica del mismo (muchas veces es muy difícil entender que se quiso hacer en un determinado momento).

Se especifica por tanto el formato de los comentarios javadoc así como ejemplos de los mismos para los casos en que se fijan para un método nuevo o se modifican para un método que ya tenía pero ha sufrido modificaciones en su lógica. Se distinguen por tanto, distintas versiones con sus correspondientes autores para cada uno de los métodos.

6. La última parte de este primer bloque del proyecto ha sido la corrección de bugs que se detectaron en 2 fases distintas:

    a. Una primera fase de revisión del código en el orden en que se ejecutaba a la hora de arrancar AGM, crear un jugador y conectar un jugador.
    b. Una segunda fase de corrección de bugs que se apreciaban a la hora de jugar al juego. La idea partió en ir corrigiendo bugs en el modo 1 jugador para luego pasar a modo multijugador aunque al final esta segunda subfase no se llevó a cabo debido al alto número de bugs en el modo 1 jugador.

    Los bugs se enumeran uno por uno en un formato fijado suficiente para comprender exactamente el origen del problema y cómo se solucionó. También en el código ha quedado esto reflejado

    Es importante mostrar cuál ha sido la destreza adquirida a la hora de corregir bugs y cómo se llevó a cabo para proseguir con el trabajo en un futuro. Se da cuenta de cuáles han sido los pasos a seguir en la mayoría de correcciones : cómo arrancar el servidor, dónde poner *breakpoints* de Eclipse para poder realizar así un depurado de la aplicación, etc…

    La mayoría de bugs que se enumeran han sido corregidos exceptuando los 2 últimos. En éstos lo que se ha hecho es dejar constancia de cuáles han sido los pasos en la investigación para dar con el origen del problema pero no se ha llegado hasta él. Estos bugs deberían ser los primeros en corregirse en un futuro y para ello se dejan entonces como abierto a la hora de buscar la reparación.

Por todos los temas que se tratan en este primer bloque, se deberían leer detenidamente y prestar especial atención a todas las partes que lo componen y comprender totalmente lo que aquí se explica para que así se esté en buenas condiciones a la hora de afrontar la continuación del desarrollo de AGM.

Una vez que se consiguió rearrancar la aplicación se puso de manifiesto que la interfaz gráfica dejaba bastante que desear y que en poco o nada se parecía a la de la primera versión del juego que, ejecutándose bajo MS-DOS, lucía mucho más intuitiva y amigable. De este modo se estableció como uno de los objetivos mejorar la jugabilidad de la AGM llevando a cabo una labor de reingeniería hasta conseguir replicar aquella primera interfaz.

Así, para la nueva propuesta (que incluiría también los paneles de bienvenida, creación del personaje y conexión) se pensó dividir la interfaz gráfica en tres partes claramente diferenciadas: la pantalla de juego en la mitad superior, un panel que agrupara la botonera de acciones, el inventario y el chat, en la parte inferior, y en medio un display en el que se mostrara la acción pretendida por el jugador.

Una vez elaborada la nueva propuesta se comenzaron a abordar los posibles cambios… Pero antes de introducir ninguno, y para evitar problemas no deseados que afectaran a la lógica del juego existente en ese momento, se hacía necesaria una comprensión total y absoluta del estado de la interfaz: saber cómo estaba estructurada, que hacía cada clase, cada método, quien les invocaba, qué función tenía cada componente, etc. Y no sólo de la interfaz sino también del modo en que ésta se relacionaba con el resto de la aplicación.

Esta tarea no fue fácil en absoluto dado que no figuraba prácticamente nada ni en la documentación de los dos años anteriores ni en el código, y respecto a la interfaz en sí, los conocimientos de AWT/Swing distaban mucho de ser los apropiados para abordar una de esta complejidad.

A esta inexistencia de información se unía lo caótico de la estructura tanto a nivel de código como a nivel visual. Por un lado clases, métodos y variables que no hacían nada, elementos redundantes, nombres no significativos, la no utilización de constantes, artificios engañosos (como el de hacer creer que era una aplicación válida para distintas configuraciones), uso inapropiado de componentes… Por otro lado, el caos en el código se plasmaba a nivel visual en una interfaz desordenada e irregular.

El siguiente paso fue por tanto tratar de entender lo que hacía cada clase, cada método y cada variable, añadiendo comentarios de línea, haciendo trazas, dibujando esquemas y bocetos, apuntando posibles mejoras, etc.

Así, se pudo llegar a una idea más o menos clara de cómo estaban organizados los componentes y a evidenciar lo que ya se sospechaba: el uso innecesario de algunos componentes (especialmente el apilamiento de contenedores **Box**) y el mal uso de otros (por ejemplo utilizar un **JButton** para mostrar una imagen pudiendo utilizar **JLabel**).

Partiendo de esto, y con la idea clara de la interfaz que queríamos conseguir se comenzó una labor de reingeniería de manera gradual realizando capturas y guardando versiones "seguras" cada vez que se solucionaban los problemas que se iban produciendo a la hora de jugar.

Tras este lento y laborioso proceso, se consiguió una interfaz gráfica más limpia y ordenada tanto a nivel de componentes como a nivel visual y que se haya dividida en tres partes claramente diferenciadas:

1. Los dos tercios superiores se han destinado a la pantalla de juego propiamente dicha donde se muestran los distintos objetos y personajes que existen en cada habitación.

2. El tercio inferior queda reservado a la botonera de las acciones, el inventario y el chat.

3. Separando ambas partes aparece un display en el que se muestra el objeto seleccionado o la acción que pretende realizar el jugador.

Esta nueva distribución de los componentes obedece a la interfaz de la versión MS-DOS a la que se hacía referencia al principio. Distribución que se buscaba replicar y a la que se ha incorporado el chat para la versión multijugador que existía en las versiones bajo Linux de años anteriores.

La pantalla de juego y la consola se han ajustado en anchura al tamaño máximo del panel que las contiene. El escenario del juego se ha agrandado en altura hasta ocupar exactamente las dos terceras partes superiores de la pantalla tal y como aparecía en la versión MS-DOS.

En cuanto al display se han agrupado todos los componentes (objeto seleccionado, caja de frase y botón Ejecutar) en un único panel y se ha situado, como en la versión MS-DOS, entre la pantalla de juego y los componentes del panel inferior.

Siguiendo con el ejemplo de la versión MS-DOS, la botonera de acciones se ha ampliado y situado en la esquina inferior izquierda de la pantalla. Las imágenes se han agrandado y, para algunas acciones, se han buscado unas más significativas y acordes con el resto de la interfaz. Así mismo, en lugar de un *tool tip* se ha acompañado a las imágenes del nombre de la acción para facilitar la jugabilidad.

Del mismo modo, el inventario, antes situado arriba en el interior de un incomprensible JSplitPane, aparece ahora junto a la botonera de acciones en la parte inferior de la pantalla. En la búsqueda de una mayor limpieza de la interfaz, la etiqueta que acompañaba al objeto, y que parecía redundante, ha sido sustituida por un *tool tip* que muestra el nombre del objeto al situarse encima el puntero del ratón.

En cuanto al chat, también se ha cambiado su ubicación situándolo en la esquina inferior derecha, para reservar así la parte central y superior a la pantalla de juego. Además se ha cambiado la imagen del botón por una mayor y más acorde con el resto de la interfaz, se han centrado los paneles entrante y saliente y se han separado un poco más. A las mejoras estéticas hay que añadirle una funcional que no es otra que la incorporación de un *scrollbar* horizontal y uno vertical que sólo serán visible cuando se necesite por la longitud del mensaje o por el tamaño de la conversación.

En cuanto a los paneles de bienvenida, creación del personaje y conexión, se han incorporado imágenes, botones, fondos y se ha jugado con colores, tamaños, fuentes y otros detalles hasta conseguir un aspecto más atractivo para el jugador.

A la sustitución de algunas imágenes por otras más idóneas en el conjunto de la interfaz, hay que añadir la unificación de criterios en las restantes y la introducción de optimizaciones en cuestiones tales como el formato y la resolución.

A las mejoras a nivel visual hay que sumar las mejoras en cuanto al código. Las líneas de código han disminuido drásticamente al utilizarse sólo las clases, métodos, variables y componentes estrictamente necesarios, llegándose a eliminar hasta cinco clases, decenas de métodos, variables y componentes y cerca de mil líneas de código que o bien no servían para nada o bien se podían optimizar. Todo ello da cuenta de la situación encontrada al abordar la interfaz original, agravada como ya se ha comentado por la inexistencia de comentarios y documentación

El resultado es una interfaz mucho más limpia y eficiente tanto visualmente como a nivel de código, que será más fácil de comprender y modificar en un futuro (por ejemplo para terminar de integrar la animación) pues se comenta el código adecuadamente (JavaDoc) y se explica todo detalladamente en esta documentación.

En esta documentación se explica el proceso de reingeniería a dos niveles: por un lado a nivel de código (analizando los cambios en las clases y métodos involucrados) y por otro a nivel visual (analizando los cambios en cada uno de los componentes).

Tras comprobar que la documentación sobre todo lo relacionado con la interfaz gráfica era casi inexistente y que ni las clases ni los métodos involucrados aparecían comentados se estableció un segundo objetivo: crear dicha documentación y comentar el código adecuadamente de tal forma que la interfaz gráfica, y el modo en que se relaciona con el resto de la aplicación, fuera más fácil de comprender y modificar en un futuro (cabe señalar que esta idea de facilitar la labor a nuestros sucesores ha sido predominante en nuestro proyecto dados los problemas con que nos encontramos desde un principio).

Así, en la presente documentación se estudia la interacción del usuario con la aplicación mediante diagramas de secuencia y flujos de sucesos de los diferentes casos de uso: por un lado para arrancar el sistema, crear un personaje y conectarlo, y por otro, una vez creada la partida, para ejecutar una acción (ir, coger, usar o mirar), para hablar con un personaje o para hablar con el resto de personajes conectados.

Para facilitar la comprensión de estos casos de uso se incluye un análisis de la estructura básica con J2EE de nuestra aplicación, comentando la arquitectura tanto del lado del usuario (ClienteAdministrador y ClienteJugador) como del lado del sistema (GestorSistemaBean, KernelBean, OyenteBean, HabitacionBean, Queue/COLA_KERNEL, Topic/CONFIRMACION y Topic/idHab)

El tercer objetivo sería continuar con la integración de la versión animada que se empezara en el curso 2004/05, incluyéndose un estudio de su situación actual tras los cambios acometidos en la interfaz.

En este sentido y, dados los problemas para revivir la aplicación, los surgidos durante la reingeniería de la interfaz y el tiempo que conllevaron el entendimiento global de la aplicación y su documentación, los avances en la integración de la animación han sido menores de lo deseado.

Lo que se ha conseguido en este punto es integrar en la nueva interfaz los paquetes con las clases modificadas hace dos cursos con lo que siguen pendientes de implementación los mismos métodos que quedaban entonces pero ahora sobre una interfaz más fácil de entender.

Para finalizar, las mejoras que no han podido llevarse a cabo por falta de tiempo pero que podrían acometerse en el futuro han sido incluidas en un último apartado. Dichas mejoras hacen referencia al problema de las distintas configuraciones, a los caracteres especiales, a las comunicaciones entre un jugador y un personaje, a la supresión del botón Ejecutar y a otros aspectos relacionados con la mejora de la jugabilidad.

Debido a todas las mejoras realizadas en la aplicación decidimos también el crear un editor de misiones para la aplicación.
Nuestra intención a la hora de desarrollar el creador de historias para AGM es que personas sin ningún tipo de conocimiento sobre la aplicación e incluso sin conocimientos de programación puedan crear sus propias misiones de manera sencilla y eficiente. Además permite un mayor control sobre las mismas al poder tener cada una de ellas en un fichero independiente, que se podrá cargar posteriormente en el editor para ser modificado.

El editor se divide principalmente en tres módulos: objetos, personajes y habitaciones. Ya que las historias se construyen básicamente a partir de objetos y personajes y la interacción entre éstos en las distintas habitaciones de la aplicación, de tal forma que definiendo los estados y acciones de los objetos y personajes y las conexiones entre las distintas habitaciones tenemos definida la misión.

El editor de misiones nos permite crear nuevas misiones, cargar misiones ya existentes para modificarlas, guardar misiones e importar una misión a AGM. Al importar una misión se escribirá la ruta del fichero .agm de la misma en el fichero de texto misiones.txt que se leerá posteriormente en AGM para saber que misión se debe cargar. Tanto en crear nueva misión como en cargar misión y guardar misión se ejecutará un explorar de ficheros desde el cual podremos acceder al directorio que queramos para guardar o cargar la misión, por lo tanto no hará falta escribir la ruta de la misión a cargar o guardar a mano.

En principio no se podrá cargar más de una misión editada a la vez, sin embargo esta misión si se podrá realizar junto con el resto de misiones del juego original. Ya que podremos añadir objetos en habitaciones ya existentes en AGM y crear puertas entre habitaciones ya existentes en la aplicación original y habitaciones de nuestra misión editada. Es decir, las misiones editadas no se ejecutan a parte de las ya existentes si no que se incorporan a ellas.

El Framework está desarrollado en Java bajo Eclipse en el paquete Editor dentro del proyecto AGM, aunque también hemos necesitado modificar ciertos métodos de AGM para el correcto funcionamiento del editor de misiones y su integración con la aplicación. Para el desarrollo de los diagramas de clases y de secuencia hemos utilizado el IBM Racional Rose Modeler Edition.

El editor está diseñado como un modelo vista controlador, de tal forma que el framework está divido en paquetes. Estos paquetes son: vista, modelo, integración y controlador. Cada uno de los cuales está subdivido a su vez en subpaquetes para los objetos, personajes y habitaciones.

La capa vista consiste en interfaces graficas Java desde las cuales interactuamos con los datos de la misión y realizamos las distintas acciones disponibles. Tanto la interfaz gráfica principal (EditorGUI) como las generales de objetos (ObjetosGUI), personajes (PesonajesGUI) y habitaciones (HabitacionesGUI) implementan la interfaz IGUI que contiene el método *actualizar (int evento, object datos)* el cual permite que las interfaces principales manden eventos al controlador.
Para cada módulo del editor hemos creado su propio grupo de guis con las opciones disponibles para cada elemento. De esta forma los subpaquetes del paquete vista y las clases que contienen son:

Objetos – ObjetosGUI, NuevoObjetoGUI, ModificarObjetoGUI, CoordenadasObjetoGUI

Personajes – PersonajesGUI, NuevoPersonajeGUI, ModificarPersonajeGUI, CoordenadasPersonajeGUI, NuevaConversacionGUI, ModificarConversacionGUI

Habitaciones – HabitacionesGUI, NuevaHabitacionGUI, ModificarHabitacionGUI, CoordenadasPuertaGUI

La clase principal de cada interfaz gráfica nos mostrará según la cargamos el primer elemento del tipo de la interfaz, si es que lo hay, y nos permitirá: crear un nuevo elemento, modificar el elemento que se está mostrando actualmente, eliminar el objeto que se está mostrando, mostrar los datos del siguiente objeto y mostrar los datos del objeto anterior.

Las clases del tipo Nuevo y Modificar como su propio nombre indica nos permitirán introducir los datos necesarios para crear un nuevo elemento o modificarlo respectivamente, teniendo en cuenta que no se pueden crear varios elementos con el mismo nombre. Mientras que las clases del tipo Coordenadas mostrarán una ventana con una imagen (por ejemplo, en el caso de CoordenadasObjetosGUI se nos mostrará la imagen de la habitación en la que hemos elegido que esté el objeto) y al clickear sobre cualquier parte de la misma devolverán las coordenadas de la posición en la que hemos clickeado, esto nos servirá para obtener de forma sencilla y rápida las coordenadas de la posición de un objeto en una habitación, de las puertas, así como de los personajes.

Al igual que al cargar las misiones se nos mostraba un explorador de archivos también se mostrará en el caso de las imágenes que necesitemos para crear objetos, personajes o habitaciones. En este caso, hemos creado un filtro para el explorador de tal forma que solo considere correcta la elección del fichero si es del tipo .jpg para las habitaciones, del tipo .gif para los objetos y del tipo .png para los personajes; ya que estos son los formatos que se usan en la aplicación original.
Además hemos creado la clase PreviewImagen que extiende JComponent e implementa PropertyChangeListener gracias a la cual obtendremos un thumbnail de los ficheros de tipo imagen en el explorador de ficheros para facilitarnos la elección de las imágenes.

Al ejecutar una acción se lanza un evento desde el gui en el que se realizó la acción al Controlador el cual a su vez delega en la Fachada, con su correspondiente interface Fachada y la clase que la implementa FachadaImp, en la que están definidas todas las acciones posibles y las clases encargadas de gestionar cada una de ellas. Estas clases son GestiónObjetos, GestiónPersonajes y GestiónHabitaciones que gestionan los métodos relacionados con los objetos, personajes y habitaciones respectivamente.
Las clases de gestión a su vez, delegan en el paquete integración para acceder al fichero de la misión y de esa manera obtener los datos que necesitan para realizar las acciones. Las clases del paquete integración son DaoObjetos, DaoPersonajes y DaoHabitaciones cada una de las cuales se encarga de gestionar el acceso a los objetos, personajes y habitaciones respectivamente, las tres clases implementan la interfaz interfaceDAO en la que se definen los métodos comunes a los tres Daos:

- *Transfer dameDato (Long id) throws IOException;*

- *Long nuevoDato (Transfer datos) throws IOException;*

- *boolean modificaDato (Transfer datos) throws IOException;*

- *void borraDato (Long id) throws FileNotFoundException, IOException;*

- *int ultimoElemento() throws IOException;*

Una vez realizada la acción el resultado obtenido se devuelve hasta el Controlador, el cual según el valor del resultado lanza un evento hasta la interfaz gráfica (EditorGUI) principal y ésta lanza de nuevo el evento hasta la interfaz gráfica original en la que se realizó la acción para que se actualice de acuerdo con el resultado obtenido.

Cabe destacar también que hemos utilizado Factorías para la creación de los objetos, estas factorías son FactoríaDaos, FactoríaGuis, FactoríaLogica y FactoríaTransfers; que se encargan de crear los objetos de tipo dao, los objetos de tipo gui, los objetos de gestión de acciones y los transfer respectivamente.
Los eventos están definidos dentro del paquete vista en la clase Eventos como valores de tipo int estáticos y públicos, de tal forma que se pueda acceder a ellos directamente sin necesidad de métodos de acceso.

Las misiones se almacenan como ficheros de extensión .agm, estos consisten en la escritura a un fichero de la clase Mision.java que implementa la clase Serializable, de tal forma que se puede escribir y leer directamente como objeto. La clase Misión consta de un String con el nombre de la misión y tres vectores que contienen los objetos, personajes y habitaciones que constituyen la misión.
La clase misión por lo tanto, tiene métodos para acceder directamente a estos vectores y poder modificarlos directamente sin necesidad de devolver el vector correspondiente en cada ocasión.

Los objetos se definen en la clase TransferObjeto que implementa también Serializable por las mismas razones que la clase Misión. La clase TransferObjeto servirá para pasar la información de cada uno de los objetos por las distintas capas y eventos del editor. Esta clase contiene toda la información necesaria para crear un objeto en AGM, desde los estados que tiene el objeto, la habitación en la que está, su posición dentro de la misma, etc... Tanto los personajes como las habitaciones están definidos de la misma forma, usamos TransferPersonaje para los personajes y TransferHabitacion para las habitaciones, estos transfer contienen, al igual que con los objetos, los datos necesarios para poder crearlos posteriormente en la aplicación principal AGM.
También tenemos TransferConversacion y TransferPuertas que se usarán como atributo en forma de vector de transfer en TransferPersonaje y TransferHabitacion respectivamente.
Las cinco clases de tipo Transfer implementan también la interface Transfer.

Una vez tenemos la misión creada como un fichero .agm si queremos jugar esa misión en la aplicación principal tendremos que clickear en la opción Importar Misión de la interfaz gráfica principal del editor. Esta opción generará un fichero con la ruta en la que se encuentra la misión generada que leerá posteriormente AGM para saber la misión que debe cargar, si es que tiene que cargar alguna. Y a su vez, está opción también copiará todas las imagenes de los objetos, personajes y habitaciones de la misión al directorio “imagenes” de AGM y las renombrará de acuerdo con los formatos de lectura de imágenes de la aplicación principal AGM, esto es: a un objeto de nombre Nom le corresponde en el estado Est la imagen Nom_Est.gif y a una habitación cuyo id es Id le corresponde la imagen HId.jpg
El motivo de copiar y renombrar las imágenes de la misión editada de acuerdo con el formato de la aplicación principal es porque intentamos que el editor afecte lo menos posible al código de AGM, al estar trabajando los tres miembros del proyecto a la vez sobre distintos aspectos del mismo.

En cuanto al trabajo y modificaciones en el proyecto original AGM, cabe destacar que en el estado actual el proyecto está aun abierto a posibles modificaciones que vayan surgiendo al terminar de implementar las opciones del editor, sobre todo teniendo en cuenta que el trabajo a realizar por otro compañero es una modificación en el apartado gráfico que podría tener repercusión sobre algunas opciones del editor como la captura de coordenadas o las propias imágenes.
Por lo tanto las modificaciones que se citan a continuación son las que ya se han tenido que realizar.

En el proyecto AGM original al crearse los Objetos y Personajes a mano se guardaban cada uno en su propia clase .java, al automatizarse esta tarea con el editor no es posible crear una clase para cada objeto por lo que hemos necesitado añadir dos clases nuevas: ObjetoEditor y PersonajeEditor.
Estas clases son genéricas para todos los objetos y personajes de la misión editada respectivamente, es decir, todos los objetos creados por una misión editada serán del tipo ObjetoEditor y todos los personajes del tipo PersonajeEditor.
Las habitaciones no tienen problemas en este sentido puesto que se crean directamente en el método creaMundo() de GestorSistemaBean.

La clase ObjetoEditor es una ampliación de la clase Plantilla_ObjetoNoPeronaje ya existente en el proyecto y que extiende a la clase ObjetoNoPersonajeClase.
Consta del código común a todos los objetos del juego, es decir: cambio de estados, comprobación de acciones, etc... pero completando automáticamente el código que se debería añadir a mano gracias a todos los datos que se obtuvieron en el editor, como por ejemplo: nombre de los estados, texto que se debe decir al mirar el objeto en cada estado, si se puede coger o no el objeto, que objeto se coge si se puede coger, con que otros objetos o personajes se puede usar el objeto, etc.…

La clase PersonajeEditor es igual que ObjetoEditor pero con los personajes, es una ampliación de la clase Plantilla_PersonajeNoJugador que extiende a la clase PersonajeNoJugadorClase. E igual que con la clase ObjetoEditor, consta del código común a todos los personajes junto con los datos obtenidos en el editor.

Por último, hemos tenido que añadir un método en la clase GestorSistemaBean de AGM que se ejecuta dentro del método creaMundo () y en el que se lee el fichero de la misión que hemos importado desde el editor y se crean las habitaciones, objetos y personajes a partir de los vectores de transfers almacenados en la misión.
También hemos tenido que modificar el método creaMundo () para comprobar al crear las habitaciones originales del juego si tenemos algún objeto o personajes en la misión que vaya a alguna de estas habitaciones y añadirlo a estas habitaciones en caso afirmativo.

2. Trabajo futuro…

2.1. Introducción

La idea de todo lo relacionado con esta parte del proyecto surgió al repetirse en los 2 últimos años del proyecto los mismos problemas al intentar instalar el entorno necesario para el desarrollo de AGM.
En los años anteriores se dedicaron a desarrollar AGM en un entorno Linux por lo que existen 3 opciones a priori: desarrollar en Linux en nuestras máquinas, trabajar mediante conexión a servidor remoto de *GRASIA*, trasladar el entorno de desarrollo a un entorno Windows.
En el último se año, se probó a instalar el entorno en multitud de versiones Linux en máquinas propias: Mandriva, Red Hat, Suse, etc… y todas se descartaron para la ejecución del proyecto por distintos motivos. Quizá el más importante fue el caso de Mandriva en el cuál se consiguió instalar todo el entorno de desarrollo pero el problema vino al intentar arrancar la aplicación dentro de Eclipse ya que con el entorno totalmente configurado y con la "supuesta" última versión de AGM, Lomboz generaba un error de parseo en *GestorSistemaBean.java* . Sí se consiguieron arrancar versiones anteriores de AGM que no incluían ni las pruebas que se añadieron en el año 2004-2005 ni otras últimas novedades.
Este suceso hizo que se descartara el acceso remoto ya que estaríamos ante el mismo problema por lo que se buscó en el entorno Windows la versión de Eclipse y Jboss que se utilizaba en Linux y se instalaron los fuentes de la última versión y se solucionó desde aquí. Esto se decidió porque todos los componentes del grupo de SSII de este año estaban más familiarizados con este entorno por lo que se ganaría tiempo en el futuro.
Después de numerosos intentos descargando distintas versiones de Eclipse, Lomboz y Jboss se dio con la versión en la cual obteníamos el mismo error de parseo xdoclet que en Mandriva. No se consiguió averiguar cuál fue el motivo del error de parseo de los comentarios xdoclet en los que se basa Lomboz.
Lo que se hizo es investigar otras versiones antiguas de AGM y analizar los comentarios de la misma clase para la cuál no se estaban parseando los comentarios y por lo que no se estaban generando las clases para el bean GestorSistemaBean. Entonces se tuvo "suerte" puesto que en una versión anterior, el código propiamente dicho era exactamente igual en otra versión de AGM y copiando los comentarios de este conseguimos que se parseara GestorSistemaBean.
Pues bien, en este apartado se indicarán los pasos a seguir para **la correcta instalación en Windows** para así dar la flexibilidad que no se ha tenido en años anteriores en lo referente a la implantación de todo el entorno de desarrollo.

Observando el código de AGM se llegó a la conclusión de que el mismo necesitaba una reestructuración y comentarios para que se pudiera mantener el código correctamente. Esto en todo proyecto es una pieza esencial y en AGM la **mantenibilidad** del código brillaba por su ausencia.

Pensando de nuevo en facilitar la tarea en siguientes años y agilizar el desarrollo de AGM se ha propuesto un estilo de documentación que debería ser seguido en los sucesivos años para que el código sea entendible de un solo vistazo y no perder tiempo y tiempo en entender un solo método.
Se han seguido también una serie de buenas prácticas a la hora de programar que se enumeraran con ejemplos y que también deberían seguirse a lo largo del desarrollo de AGM.
El sistema de trazas de log4j se ha añadido en el año 2006-2007 a AGM y se explicará su potencial y su uso dentro del proyecto para aprovechar así todas las ventajas que ofrece sobre el antiguo *System.out.*
También se siguió el tutorial de Sun para el repaso de los métodos de los distintos Beans de AGM y se puso especial énfasis en los métodos finder, create, store y load del EntityBean HabitacionBean. En la versiones anteriores no se seguían las normas que se marcan en el tutorial de Sun de J2EE sobre el formato de estos métodos y sobre qué deben hacer en ciertos casos. Se han lanzado las excepciones que espera la arquitectura en cada momento para que el contenedor las gestione como se espera.

También se repasó el juego y se han ido corrigiendo ciertos **bugs** que ocurren a la hora de jugar a AGM. Se empezó en modo monojugador y se ha continuado por ahora, dejando el modo multijugador para cuando este primero funcione.
Se explicará la forma de proceder a la hora de detectar ciertos bugs para que en un futuro se prosiga con esta labor o se aborde de otra forma que se ocurra.
Se explican los bugs en un formato adecuado para su comprendimiento.

2.2. Preparándose para el entorno…

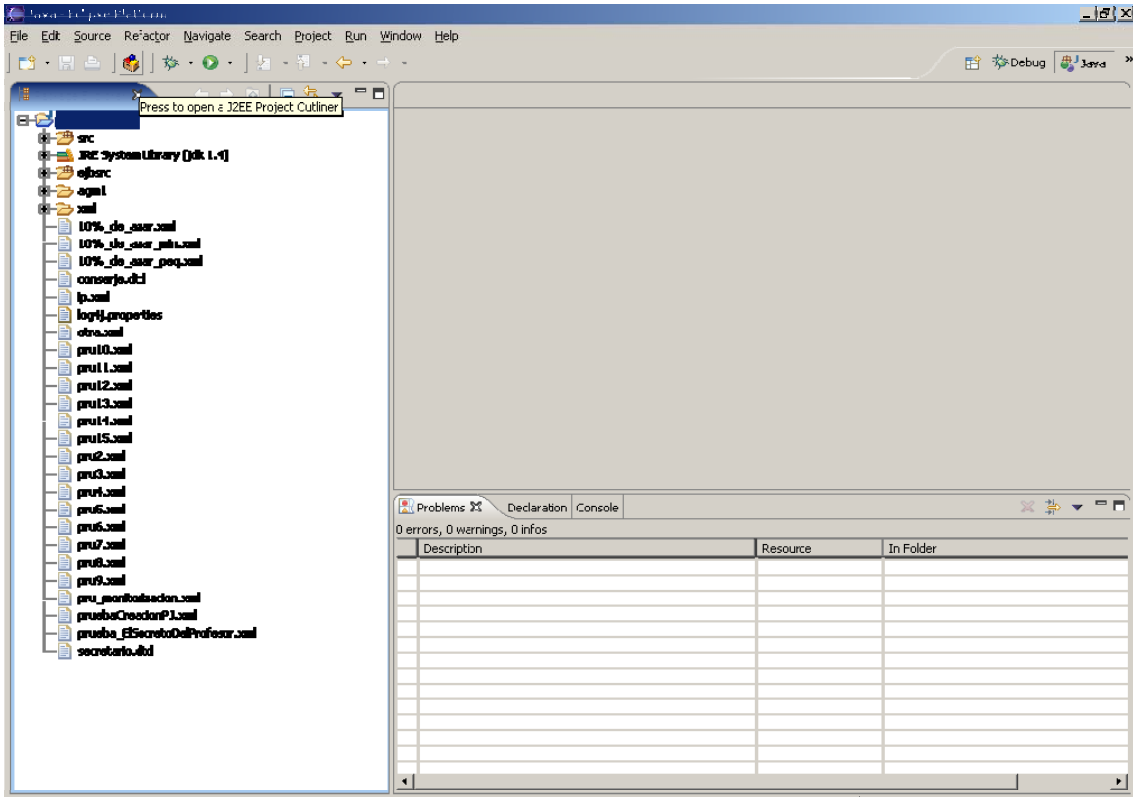
Todo miembo del proyecto de AGM debería tener conocimientos suficientes para poder afrontar el desarrollo del mismo.
AGM es un proyecto J2EE que se ejecuta sobre un servidor de aplicaciones. Se da por hecho que los integrantes del grupo tienen los conocimientos necesarios de Java como para afrontar un proyecto J2EE. La JDK utilizada será la 1.4.2
Por tanto lo primero antes de nada es entender bien como funciona la arquitectura J2EE (capítulo 1 del enlace siguiente). AGM se basa en un módulo EJB formado por 4 EJB´s distintos. Se propone el capítulo 23 del siguiente enlace que da cuenta de lo que es cada uno de los distintos tipos de EJB con un caso práctico e información detallada.
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

También se cree necesario conocer los detalles más relevantes del servidor de aplicaciones sobre el cuál se ejecuta AGM. Este es JBoss version 3.2.1 e información detallada se proporciona en el documento adjunto ***jbossj2ee.pdf*.**

Por último es vital manejar el entorno de desarrollo utilizado en todos estos años anteriores. Este es Eclipse 3.0 + Lomboz. El manejo de Eclipse se da por hecho y en la siguiente sección se explica por encima cómo se debe manejar Lomboz para desplegar y generar EJBs.

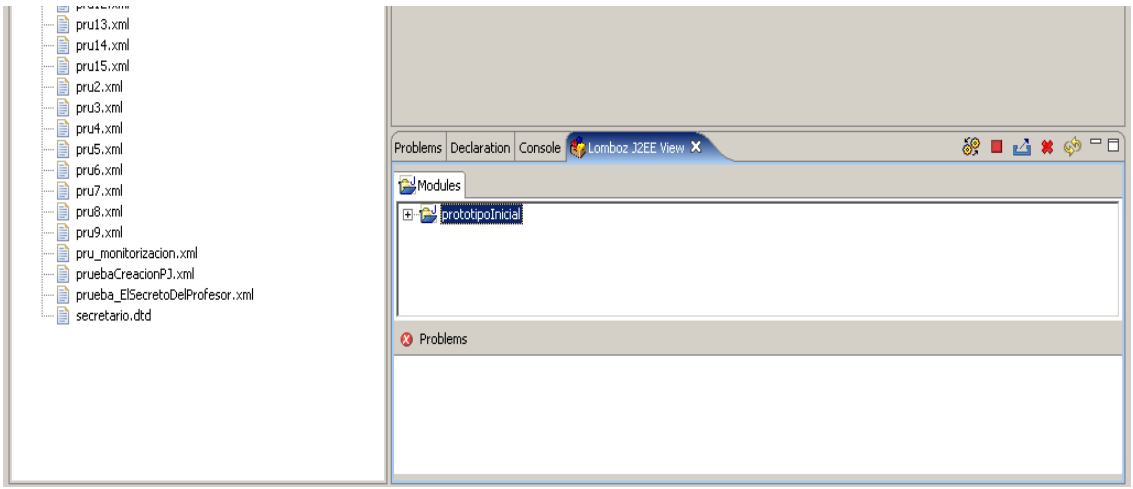
2.2.1. Lomboz

Lomboz es un plugin de Eclipse que en nuestro caso facilita toda la generación de EJB´s.
Si está bien instalado en Eclipse, debe aparecer un icono como al que apunta el cursor en la siguiente imagen



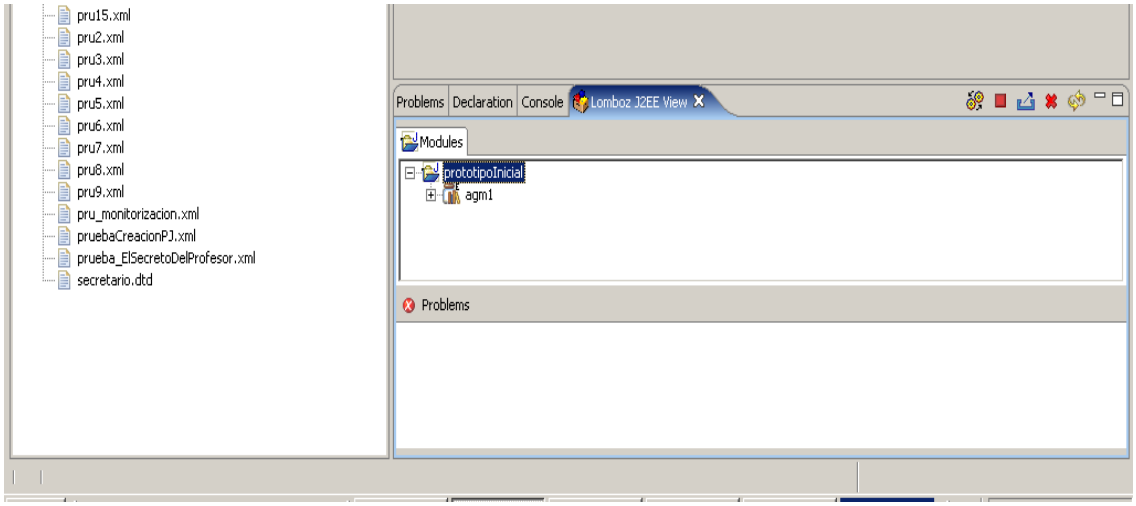
2.1 Eclipse + Lomboz

Si pulsamos en el icono de Lomboz, se abre la vista que ofrece la funcionalidad del plugin.



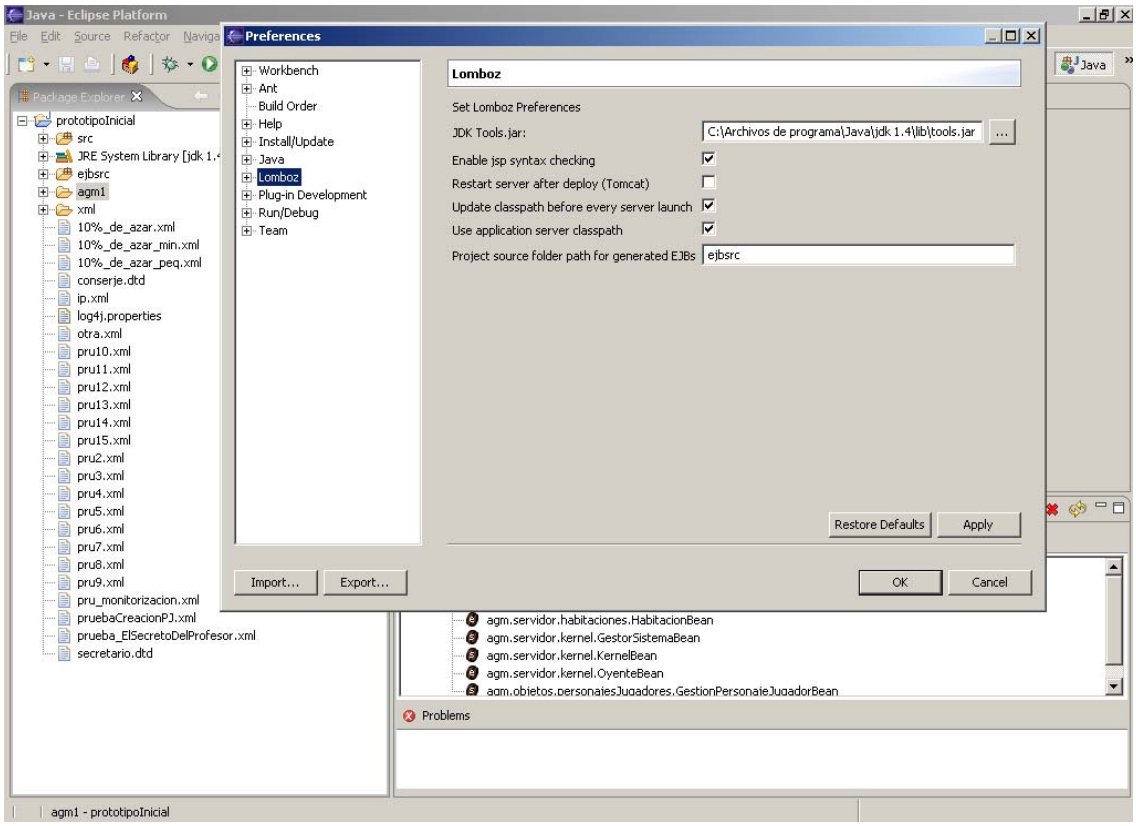
2.2 Vista Lomboz

La vista se divide en proyectos abiertos en ese instante en Eclipse. Cada proyecto puede contener varios módulos que se añaden pulsando el botón derecho sobre el proyecto. Existen 2 tipos de módulos : web y de EJB. En el caso de AGM el proyecto prototipInicial tiene un módulo EJB de nombre “agm”. Al estar creado es raro que haya que crear otro nuevo.



2.3 Módulo agm1

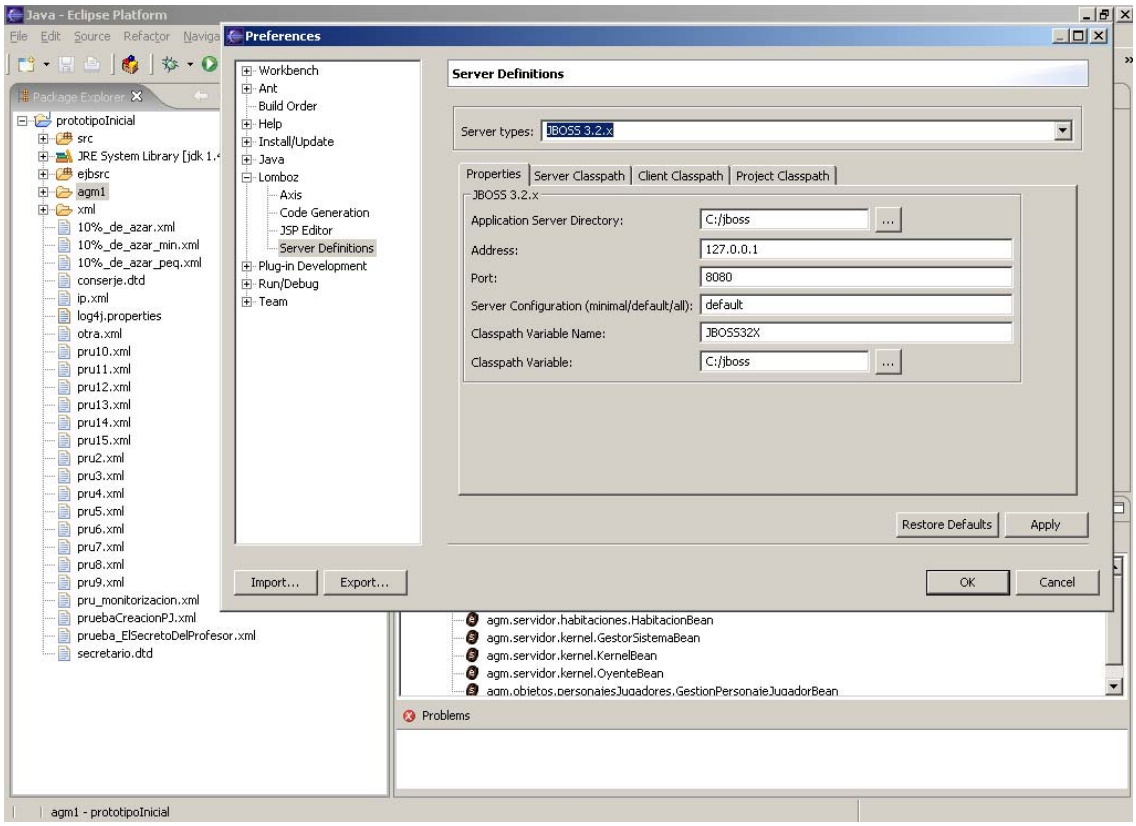
Todo módulo tiene asociado un servidor. Los distintos servidores que se pueden asociar a un módulo se especifican en el menú Windows-> Preferences -> Lomboz



2.4 Configuración Lomboz

Este es el menú que configura Lomboz, entre otras cosas se configura el jar “Tools.jar” de la jdk correspondiente necesario para el correcto funcionamiento de Lomboz.

Además se puede especificar el nombre de la carpeta que se ubicará en el raíz del proyecto y que contendrá los ficheros EJB´s que más tarde se “moverán” al servidor especificado.
Además, existen otras opciones en este menú de configuración de Lomboz entre las que se encuentra la definición de los distintos servidores que podemos usar para nuestros módulos.
En AGM sólo importa el servidor de nombre JBOSS 3.2.X
Un servidor se define entre otras cosas por el directorio donde se ubica, el puerto, la direccion IP a usar, etc…
Esto en un principio no será necesario modificarlo ya que ya se dará configurado.



2.5 Configuración servidores Lomboz

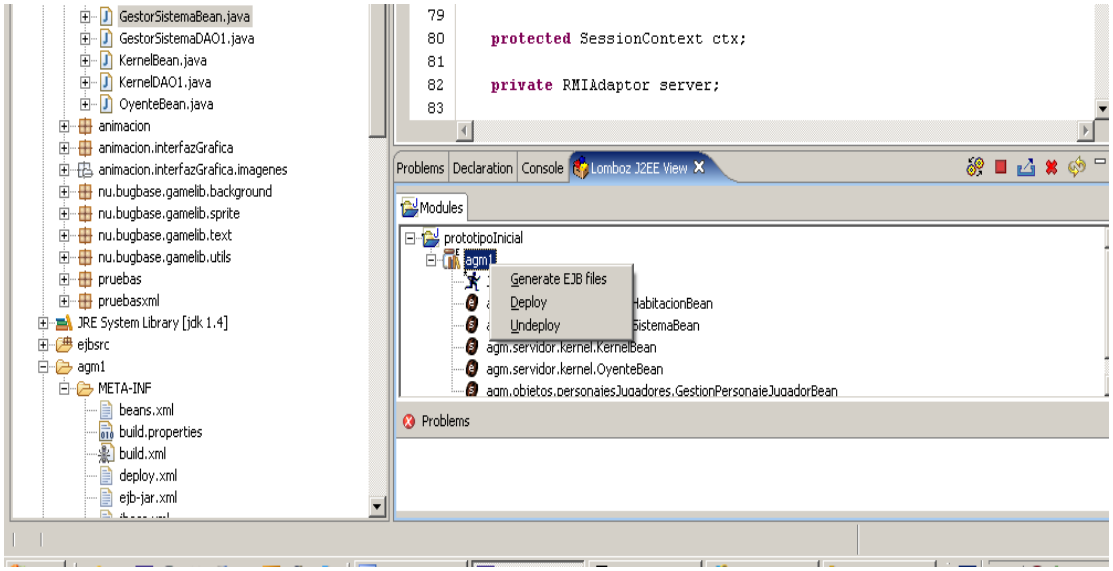
En AGM, el módulo EJB de nombre “agm” tiene asociado el servidor JBOSS 3.2.X y contiene 5 EJB´s:

- HabitacionBean, BMP
- GestorSistemaBean, es un SessionBean.
- KernelBean, es un SessionBean.
- OyenteBean, es un MDB.
- GestionPersonajeJugadorBean, este último Bean se podría obviar ya que no aporta nada.

La forma de trabajar con Lomboz es la siguiente. Se codifica en el paquete src del proyecto prototipoInicial los métodos de los EJBs. En el caso de que se estén añadiendo métodos que se invoquen remotamente desde una aplicación cliente, hay que añadir el método pulsando con el botón derecho sobre el EJB en la vista de Lomboz y clickeando sobre “Add method”. En el caso de que el método sea privado, esto no será necesario.

Tan sólo habría que añadirlo como un método más a nuestro fichero java correspondiente.
En el módulo EJB existen varias opciones al pulsar sobre él con el botón derecho:

- "Generate EJB Files", esta opción lo que hace es parsear los comentarios xdoclet de los distintos EJBs de nuestro módulo y generar las clases necesarias en "ejbsrc" para formarlos. Es aquí donde se ve claramente la ayuda que ofrece Lomboz en el desarrollo J2EE.
- Deploy, significa empaquetar el paquete "ejbsrc" y transformarlo en un jar que se copiará al directorio del servidor asociado al módulo. El despliegue en Jboss es en caliente. Esto quiere decir que en el caso de que se despliegue con el servidor encendido, se cargarán en el servidor automáticamente los cambios. El fichero que se genera se llama *"agm1.jar"*
- Undeploy, elimina el jar correspondiente del directorio del servidor.



2.6 Opciones módulo

Lo único que falta por destacar sobre la utilización del plugin de Lomboz son las distintas opciones asociadas al servidor:

- **Run Server**, arranca el servidor en modo "Run" y carga todos los módulos que pudieran existir en la carpeta "default".
- **Debug Server**, igual que el anterior pero lo arranca en modo "Debug". En este modo, es posible añadir puntos de interrupción en los ficheros del paquete src relacionados con los EJB (clases DAO, etc…) y se detendrá la ejecución de AGM y podremos depurar la ejecución en los mismos EJB (que para entonces vivirán empaquetados en un jar dentro del servidor).
- **Stop Server**, para el servidor.

La forma de proceder debería ser la siguiente:

- Se realizan modificaciones en los ficheros java del paquete src.
- Pulsamos sobre el módulo "agm" en la vista de Lomboz para generar los ficheros EJB ("Generate EJB Files") y seguidamente desplegamos el módulo ("Deploy").

- Si el servidor no estaba arrancado se puede hacer en cualquiera de los 2 modos (“Run Server” ó “Debug Server”).
- Se ejecuta cualquier clase cliente de AGM (ClienteAdministrador o ClienteJugador).

2.3. Preparando el entorno…

En el año 2004-2005 estuvieron intentando arrancar la aplicación hasta Enero del 2005.
En el año 2006-2007 consiguieron arrancar la aplicación en Febrero de 2007.
Este retraso en los 2 últimos años en los que se ha proseguido con el proyecto hacen necesario una solución. No se puede permitir que cada año se tengan los mismos problemas.
Para ello se propone dar los pasos necesarios para la instalación del entorno necesario en el cual se pueda ejecutar y desarrollar AGM.
Se proporcionarán los pasos básicos a seguir para la instalación en Windows. Se explicará y se comprobará la instalación para versiones Windows XP.
Para quién desee trabajar en Linux, se proporcionan los fuentes del proyecto (prototipo2006-2007.rar) según se dejó en el último año y las versiones Linux de jboss y eclipse+lomboz (jbossLinux.rar y eclipseLinux.rar). Los pasos son similares a los que se describen para Windows XP.

2.3.1. Instalación en Windows XP

Junto con esta documentación se entregan 2 ficheros comprimidos que contienen jboss y eclipse+lomboz+agm (jboss.rar y eclipse.rar respectivamente).
Ambos ficheros deben ser descomprimidos en el directorio raíz de C. Antes hay que asegurarse de que no existan carpetas con el mismo nombre (eclipse y jboss).
Necesitamos instalar la versión de la jdk 1.4.2. Lo haremos en *“C:\Archivos de programa\Java\jdk 1.4”* para evitar así el tener que cambiar ciertas variables luego en eclipse. En el caso de que no eligiéramos este destino habría que cambiar la jre usada por Eclipse para los proyectos y el Tools.jar de Lomboz.
Por último es necesario crear 1 variable de entorno:

- ***JAVA_HOME*** apuntando a C:\Archivos de programa\Java\jdk 1.4

De esta forma seríamos capaces de arrancar jboss desde línea de comandos desde la carpeta c:\jboss\bin mediante el fichero run.bat.
Aunque gracias a Lomboz esto no será necesario ya que Jboss se arrancará mediante el plugin como se indicó en el apartado anterior.
De esta sencilla forma se estaría en la misma disposición en la que se estuvo en los 2 últimos años en los que se prosiguió con el proyecto y que tanto esfuerzo costó y tanto tiempo se perdió.

2.4. Mantenibilidad de AGM

Se ha perseguido que el desarrollo de AGM en años posteriores sea lo más fácil posible y para ello se ha dejado el entorno bien preparado para su instalación tal y como se ha explicado en el apartado anterior.
Pero quizá uno de los aspecto más relevantes para este propósito sea el de la mantenibilidad. El ideal de la mantenibilidad estaría en una buena documentación del código y un estilo único de código que permita identificar claramente que se hace en cada momento y por qué de tal forma que antes de realizar cualquier cambio se estuviera en total disposición de evaluar el impacto que supondría en la aplicación.
El diseño de AGM no se retocó para buscar esta mantenibilidad.
El estilo de código varía de ciertos componentes a otros y esto se debe a los distintos miembros que hasta ahora han formado el proyecto de AGM. Se ha perseguido que todo el código se adapte a ciertas buenas prácticas de programación pero homogeneizar el estilo de código es una tarea casi imposible.
La documentación del código ha sido el gran ausente en AGM y se ha hecho un gran esfuerzo para que el código esté bien documentado (véase 2.4.3) gracias a un formato de javadoc establecido para que se mantenga en el futuro. Este permite que todo método sea comprendido incluso antes de mirar el código. De esta forma se agiliza mucho la comprensión del código y se está entonces en disposición absoluta de realizar cualquier modificación sobre AGM.

2.4.1. Log4J

Se ha adoptado el sistema de trazas de **log4j**.

Log4j es una librería open source desarrollada en Java por la Apache Software Foundation que permite a los desarrolladores de software elegir la salida y el nivel de granularidad de los mensajes o “logs” (logging) a tiempo de ejecución y no a tiempo de compilación como es comúnmente realizado. La configuración de salida y granularidad de los mensajes es realizada a tiempo de ejecución mediante el uso de archivos de configuración externos. Log4J ha sido implementado en otros lenguajes como: C, C++, C#, Python, Ruby, y Eiffel (http://es.wikipedia.org/wiki/Log4j).

Se da a continuación una explicación del manejo de log4j con el cual se mostrarán los aspectos esenciales de esta útil librería de Apache.

Log4j consta de varios **niveles** en los cuales se priorizan los mensajes y se filtran mediante un fichero de configuración que se mostrará más adelante.
Estos distintos niveles son:

- **DEBUG**: se utiliza para escribir mensajes de depuración, este log no debe estar activado cuando la aplicación se encuentre en producción.
- **INFO**: se utiliza para mensajes similares al modo "verbose" en otras aplicaciones.
- **WARN**: se utiliza para mensajes de alerta sobre eventos que se desea mantener constancia, pero que no afectan el correcto funcionamiento del programa.
- **ERROR**: se utiliza en mensajes de error de la aplicación que se desea guardar, estos eventos afectan al programa pero lo dejan seguir funcionando, como por

ejemplo que algún parámetro de configuración no es correcto y se carga el parámetro por defecto.
- **FATAL**: se utiliza para mensajes críticos del sistema, generalmente luego de guardar el mensaje el programa abortará.
- **ALL**: este es el nivel más bajo posible, habilita todos los logs.
- **OFF**: este es el nivel más alto posible, deshabilita todos los logs.

Además en Log4j se habla de **Appenders** y **Layouts**.

Los **Appenders** son los distintos destinos a los que pueden ir los distintos logs de la aplicación. Entre ellos se encuentran:
- A un **fichero** de log conocido como FileAppender o RollingFileAppender (en el caso de que se quiera rotar entre varios con el mismo prefijo para no tener un fichero de logs demasiado extenso).
- A la **consola** como el típico System.out con ConsoleAppender.
- A una dirección de **correo electrónico** con SMTPAppender
- A una **base de datos** con JDBCAppender.

Layout es el responsable de dar un formato de presentación a los mensajes. Es decir, qué forma se quiqere que los mensajes tengan (inclusión de la clase en la que se encuentra el mensaje, hora, dia, mes, año, etc..)
Además permite presentar el mensaje con el formato necesario para almacenarlo simplemente en un archivo de texto *.log* (*SimpleLayout* y *PatternLayout*), en una tabla HTML (*HTMLLayout*), o en un archivo XML (*XMLLayout*).

Para información más detallada sobre Layouts y Appenders visitar
http://logging.apache.org/log4j/docs/manual.html

¿Cómo se usa log4j?

- En toda clase (clase.class por ejemplo) que se quieta usar log4j deberemos instanciar un atributo estático de tipo Logger de la siguiente forma:
  private static Logger log = Logger.getLogger(nombreClase.class);
- Introducir las trazas del nivel adecuado como por ejemplo:
  log.debug(“Paso por aquí”);

¿Cómo se configura log4j?

Los distintos Layout, Appenders y prioridades del sistema de trazas se configuran mediante un fichero **xml** o un fichero **properties**.
En este pequeño ejemplo la demostración se basa en el fichero properties de configuración dejando al lector informarse sobre el formato del fichero xml.

El fichero properties consta de pares clave=valor. El que se usa en AGM tiene esta forma:

```
# Set root category priority to INFO and its only appender to CONSOLE.
log4j.rootCategory=DEBUG, CONSOLE
#log4j.rootCategory=INFO, CONSOLE, LOGFILE

# Set the enterprise logger category to FATAL and its only appender to CONSOLE.
#log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE
```

```
# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%d{yyyy/MM/dd HH:mm:ss.SSS}] %c %m%n

# LOGFILE is set to be a File appender using a PatternLayout.
log4j.appender.LOGFILE=org.apache.log4j.FileAppender
log4j.appender.LOGFILE.File=axis.log
log4j.appender.LOGFILE.Append=true
log4j.appender.LOGFILE.Threshold=INFO
log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
log4j.appender.LOGFILE.layout.ConversionPattern=%d %-5p %c{2} %m%n
```

Aquí se está configurando log4j de tal forma que se imprimen mensajes de nivel debug o más altos en consola. La consola está configurada para que el patron del mensaje sea el especificado por [%d{yyyy/MM/dd HH:mm:ss.SSS}] %c %m%

Si se descomentara la tercera línea y se comentara la segunda, log4j quedaría configurado de tal forma que los mensajes que se imprimieran serían de nivel INFO o superior y se haría sobre la consola y sobre fichero de tal forma que este tiene activado un filtro (Threshold) también de INFO y el formato viene especificado por %d %-5p %c{2} %m%n. Además el fichero tendrá el nombre axis.log.

¿Cómo se configura desde código?

Una vez instanciada la variable como se ha explicado anteriormente tenemos varias opciones:

- Añadir una línea que configure la variable Logger con cierta configuración.
  PropertyConfigurator.configure("log4j.properties");
  De esta forma se busca el fichero log4j.properties en el raíz del proyecto.
- No añadir ninguna línea y por lo tanto existen varias opciones: si se configuró anteriormente en la ejecución alguna variable Logger, se heredará dicha configuración; si no se hizo nada, se heredará una configuración por defecto.
- Forzar que se use la configuración por defecto mediante BasicConfigurator.configure();

¿Cómo utilizar log4j en AGM?

Existen 2 clases principales en AGM: ClienteAdministrador y ClienteJugador. Lo deseable sería configurar log4j una vez por ejecución y por eso se han añadido una línea que busca el properties en el raíz del proyecto en cada main.
Entonces el problema es el siguiente: toda ejecución de AGM tiene una parte que se ejecuta en el cliente y una parte que se ejecuta en el servidor (toda la relacionada con EJBs). La parte que se ejecuta en cliente se puede controlar en cuánto a lo que se refiere a la configuración de log4j. El problema viene con la parte servidora ya que los logs aquí dependen de la configuración que se adopte en el servidor. Esta configuración está en formato xml y se encuentra en C:\jboss\server\default\conf\log4j.xml
Entonces se tienen para las 2 ejecuciones, 2 configuraciones distintas. La configuración de jboss está activada para nivel INFO. Entonces según tenemos el properties de la parte cliente, se ven los mensajes DEBUG por consola pero no en las trazas que añadamos en los ficheros de los EJB para este nivel de prioridad.

Se hace entonces necesario establecer una forma de incluir trazas que sea común a ambas partes:

- Se deberían añadir trazas de nivel INFO en aquellos puntos donde se quiera algún tipo de información que fuera útil para el desarrollo o seguimiento de AGM.
- Se deberían añadir trazas de nivel WARN en aquellas capturas de excepciones las cuáles no sean críticas y tan solo se quiera mostrar un mensaje de advertencia por pantalla.
- Se deberían añadir trazas de nivel ERROR en aquellas capturas de excepciones que sean reflejo de un comportamiento inesperado en AGM y que provoque el mal funcionamiento del mismo.

Estableciendo este convenio podemos siempre seguir el mismo criterio tanto en la parte cliente como en la servidora y así asegurar siempre el mismo funcionamiento de nuestro sistema de trazas.

2.4.2. Buenas Prácticas

Se han adoptado ciertas buenas prácticas a la hora de repasar el código de AGM y que deberían respetarse en un futuro ya que ayudan a que el código sea más eficiente y mantenible. Se mostrarán enumerándose las distintas buenas prácticas , con ejemplos de incumplimiento de estas y con ejemplos de cumplimiento

1. Se deben cerrar recursos JDBC en bloques finally para asegurarse realmente de que no se dejan recursos abiertos ante posibles excepciones que pudieran ocurrir en los distintos métodos. Si se usa el mismo objeto PreparedStatement para varias sentencias en el mismo método, se deberían ir cerrando todos y cada uno de ellos por separado. Además hay que tener especial cuidado de no ejecutar un close sobre un recurso JDBC nulo ya que obtendremos un NullPointerException.
   Un ejemplo de incumplimiento de esta buena práctica sería el siguiente:

```
Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
     conn = jdbcFactory.getConnection();
     String queryString = "select H.idHab from habitaciones as H, situacion
as s"
               + " where H.idHab = s.idHab AND s.idO = ?";
     ps = conn.prepareStatement(queryString);
     ps.setString(1, idO);
     rs = ps.executeQuery();
     result = rs.next();
     …
} catch (Exception e) {
…
}
```

Debería ser sustituido por algo tal que así

```
Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;

try {
    conn = jdbcFactory.getConnection();
    String queryString = "select H.idHab from habitaciones as H, situacion as
s"
            + " where H.idHab = s.idHab AND s.idO = ?";
    ps = conn.prepareStatement(queryString);
    ps.setString(1, idO);
    rs = ps.executeQuery();
    result = rs.next();
    …
} catch (Exception e) {
…
}
finally {
    if (null != conn) {
        conn.close();
    }
    if (null != ps) {
        ps.close();
    }
    if (null !=rs) {
        rs.close();
    }
}
```

2. Se debe evitar en la medida de lo posible instanciar nuevos objetos en cuerpos de bucles for ya que esto degrada el rendimiento de nuestra aplicación al instanciar en cada pasada un nuevo objeto en la pila. Se pueden dar ocasiones de que esto se necesite aunque debería evitarse.
   Un ejemplo de incumplimiento de esta buena práctica sería el siguiente:

```
for (int i = 0; i < 3 ; i ++) {
    String s = new String(“incumplimiento”);
}
```

Un código correcto frente a esta buena práctica entonces podría ser el siguiente:

```
String s = new String(“”);
for (int i = 0; i < 3 ; i ++) {
    String s = “incumplimiento”;
}
```

3. No dejar cuerpos de catch vacíos. Esto es algo que no se explica como se puede cometer pero se cometía en ciertos trozos de código de AGM. Toda excepción debe ser tratada correctamente e informada por consola en la mayoría de los casos, sobre todo cuando no se controla muy bien el origen de la excepción.
   Un ejemplo de incumplimiento de la regla sería el siguiente

```
/**
     * @author Juan Carlos Castromil
     * @version 1.1 07/2007
     * Se devuelve una Collection con todos los HabitacionBean que
     * existen en este mismo instante en el servidor
     */
    public Collection findAll() throws javax.ejb.FinderException {

        List a = new ArrayList();
```

```
        Connection conn = null;
        ResultSet rs = null;
        HabitacionPK key = null;
        PreparedStatement sqlStatement = null;

        try {
            conn = jdbcFactory.getConnection();
            String sqlString = "SELECT idHab FROM Habitaciones";
            sqlStatement = conn.prepareStatement(sqlString);
            rs = sqlStatement.executeQuery();
            while (rs.next()) {
                key = new HabitacionPK(rs.getString(1));
                a.add(key);
            }
        } catch (SQLException e) {

        }
    …
```

Y se repararía por ejemplo de la siguiente forma:

```
/**
     * @author Juan Carlos Castromil
     * @version 1.1 07/2007
     * Se devuelve una Collection con todos los HabitacionBean que
     * existen en este mismo instante en el servidor
     */
    public Collection findAll() throws javax.ejb.FinderException {

        List a = new ArrayList();
        Connection conn = null;
        ResultSet rs = null;
        HabitacionPK key = null;
        PreparedStatement sqlStatement = null;

        try {
            conn = jdbcFactory.getConnection();
            String sqlString = "SELECT idHab FROM Habitaciones";
            sqlStatement = conn.prepareStatement(sqlString);
            rs = sqlStatement.executeQuery();
            while (rs.next()) {
                key = new HabitacionPK(rs.getString(1));
                a.add(key);
            }
        } catch (SQLException e) {
            throw new javax.ejb.EJBException(e);
        }
```

4. Los métodos finder de HabitacionBean deben ajustarse al patrón que se marca para este tipo de métodos en la arquitectura J2EE. Por ejemplo antiguamente había un método tal que así:

```
public HabitacionPK findByPrimaryKey(HabitacionPK pk)
            throws javax.ejb.FinderException {

        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;

        try {
            conn = jdbcFactory.getConnection();
            String queryString = "select idHab from habitaciones where idhab
= ?";
            ps = conn.prepareStatement(queryString);
            String key = pk.getIdHab();
            ps.setString(1, key);
            rs = ps.executeQuery();
            boolean result = rs.next();
            if (result) {
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
            throw new javax.ejb.FinderException(
                    "Inside HabitacionDAOImpl.findbyPrimaryKey()"
                            + "following primary key" + pk.getIdHab()
                            + "not found");
        } finally {
            try {
                if (null != ps) {
                   ps.close();
                }
                if (null != conn) {
                   conn.close();
                }
            }
            catch (SQLException e2) {
                    log.error("Error SQL al liberar recursos " +
e2.getMessage());
                    e2.printStackTrace();
            }
        }
        return pk;
    }
```

Pero el problema es que esto no se ajusta con el patrón dado como ejemplo en el tutorial del siguiente enlace sobre BMP
http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/BMP2.html#62907

Entonces se transformó el mismo para que, por ejemplo, en el caso de que no encontrara esa habitación en BD, se lanzara un ObjetcNotFoundException para que luego el contenedor tratara esta excepción. Y así con otros aspectos de tal forma que el método quedo así

```
/**
 * @author Juan Carlos Castromil
     * @version 1.0 07/2007
     * Se busca la habitación que alberga
     * al personaje no jugador indicado por idPNJ
     * @param idPNJ
     * @return HabitacionPK con el id de la habitacion que contiene
     * el personaje no jugador indicado por idPNJ
     */
    public HabitacionPK findByIdPNJ(String idPNJ)
            throws javax.ejb.FinderException {

        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        HabitacionPK pk = new HabitacionPK();

        try {
            conn = jdbcFactory.getConnection();
            String queryString = "select H.idHab from habitaciones as
H,personajesnj as p"
                    + " where H.idHab = p.idHab AND idPNJ = ?";
            ps = conn.prepareStatement(queryString);
            ps.setString(1, idPNJ);
            rs = ps.executeQuery();
            boolean result = rs.next();
            if (result) {
                pk.setIdHab(rs.getString(1));
            }
        } catch (Exception e) {
            e.printStackTrace();
            throw new javax.ejb.FinderException(
                    "Inside  HabitacionDAOImpl.findByIdPNJ()" + e);
        } finally {
```

```
            try {

                if (null != ps) {
                   ps.close();
                }
                if (null != conn) {
                   conn.close();
                }
            } catch (SQLException e2) {
                   log.error("Error SQL al liberar recursos en findByIdPNJ " +
e2.getMessage());
                   e2.printStackTrace();
            }
        }
        return pk;
    }
```

De la misma forma, se modificó el método ejbCreate de HabitacionDAOImpl. Antiguamente se encontraba de esta forma:

```
public HabitacionPK create(HabitacionBean ejb) throws CreateException,
            EJBException {

        java.sql.Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        int count = 1;
        String updateString = null;
        try {
            conn = jdbcFactory.getConnection();
            updateString = "insert into habitaciones values"
                    + "(?, ?, ?, ?, ?, ?)";
            ps = conn.prepareStatement(updateString);
            ps.setString(count++, ejb.getIdHab().trim());
            ps.setString(count++, ejb.getNombre().trim());
            if (ejb.isLuz())
                ps.setInt(count++, 1);
            else
                ps.setInt(count++, 0);
            ps.setInt(count++, ejb.getAncho());
            ps.setInt(count++, ejb.getAlto());
            ps.setString(count++, ejb.getMascara());
            ps.executeUpdate();
            String insertObjetosHabitacionString = "insert into SituacionO
values (?, ?, ?, ?)";
            String insertObjetosHabitacionString2 = "insert into objetos
values (?, ?, ?)";
            String insertPuertasHabitacionString = "insert into Comunicacion
values (?, ?, ?)";
            java.util.Iterator it = ejb.getPuertas().iterator();
            ////////////////////
            Puerta puerta = null;
            count = 1;
            while (it.hasNext()) {
                puerta = (Puerta) it.next();
                count = 1;
                ps = conn.prepareStatement(insertObjetosHabitacionString2);
                ps.setString(count++, puerta.getIdO());
                ps.setString(count++, puerta.getTipo());
                ps.setString(count++, puerta.getEstado());
                ps.executeUpdate();
                count = 1;
                ps = conn.prepareStatement(insertObjetosHabitacionString);
                ps.setString(count++, puerta.getIdO());
                ps.setString(count++, puerta.getIdHab());
                ps.setInt(count++, puerta.getPosX());
                ps.setInt(count++, puerta.getPosY());
                ps.executeUpdate();
                count = 1;
                ps = conn.prepareStatement(insertPuertasHabitacionString);
```

```
                ps.setString(count++, puerta.getIdO());
                ps.setString(count++, puerta.getIdHab());
                ps.setString(count++, puerta.getIdHabDest());

                ps.executeUpdate();

            }

        } catch (Exception e) {
        } finally {
            try {
                ps.close();
                conn.close();
            } catch (Exception e2) {
            }
        }
        return null;
    }
```

Y siempre se devolvía null por lo que se cambió también para que se adaptara al estándar J2EE tal y como marca el tutorial antes indicado. Quedó así:

```
/**
     * @author Juan Carlos Castromil
     * @version 1.1 07/2007
     * SE ha realizado una modificacion ya que antes siempre devolvía
     * null el método y ahora lo que hace es devolver el HabitacionPK
     * que encapsula el id de la habitacion
     * @param ejb es el habitacionBean que se va a crear en BD
     * @return el id de la habitacion
     */
    public HabitacionPK create(HabitacionBean ejb) throws CreateException,
            EJBException {

        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        int count = 1;
        String updateString = "";
        HabitacionPK pk = new HabitacionPK();


        try {
            conn = jdbcFactory.getConnection();
            updateString = "insert into habitaciones values"
                    + "(?, ?, ?, ?, ?, ?)";
            ps = conn.prepareStatement(updateString);
            ps.setString(count++, ejb.getIdHab().trim());
            ps.setString(count++, ejb.getNombre().trim());
            if (ejb.isLuz())
                ps.setInt(count++, 1);
            else
                ps.setInt(count++, 0);
            ps.setInt(count++, ejb.getAncho());
            ps.setInt(count++, ejb.getAlto());
            ps.setString(count++, ejb.getMascara());
            ps.executeUpdate();
            ps.close();
            String insertObjetosHabitacionString = "insert into SituacionO
values (?, ?, ?, ?)";
            String insertObjetosHabitacionString2 = "insert into objetos
values (?, ?, ?)";
            String insertPuertasHabitacionString = "insert into Comunicacion
values (?, ?, ?)";
            Iterator it = ejb.getPuertas().iterator();
            ////////////////////
            Puerta puerta = null;
            count = 1;
            while (it.hasNext()) {
                puerta = (Puerta) it.next();
```

```
                count = 1;
                ps = conn.prepareStatement(insertObjetosHabitacionString2);
                ps.setString(count++, puerta.getIdO());
                ps.setString(count++, puerta.getTipo());
                ps.setString(count++, puerta.getEstado());
                ps.executeUpdate();
                ps.close();
                count = 1;
                ps = conn.prepareStatement(insertObjetosHabitacionString);
                ps.setString(count++, puerta.getIdO());
                ps.setString(count++, puerta.getIdHab());
                ps.setInt(count++, puerta.getPosX());
                ps.setInt(count++, puerta.getPosY());
                ps.executeUpdate();
                ps.close();
                count = 1;
                ps = conn.prepareStatement(insertPuertasHabitacionString);
                ps.setString(count++, puerta.getIdO());
                ps.setString(count++, puerta.getIdHab());
                ps.setString(count++, puerta.getIdHabDest());
                ps.executeUpdate();
                ps.close();
             }
            pk.setIdHab(ejb.getIdHab());
        } catch (Exception e) {
                   e.printStackTrace();
                   throw new EJBException("Error al crear el HabitacionBean de
id " + ejb.getIdHab() + e.getMessage() );
           }
finally {
            try {
                if (null != ps) {
                   ps.close();
                }
                if (null != conn) {
                   conn.close();
                }
            } catch (SQLException e) {
                           log.error("Error al liberar recursos " +
e.getMessage());
                           e.printStackTrace();
                           throw new EJBException("Error al liberar recursos
al crear el HabitacionBean de id " + ejb.getIdHab() + e.getMessage() );
                   }
        }
        return pk;
    }
```

2.4.3. Documentación de código

La documentación de código en todo proyecto debe ser algo esencial. Las ventajas que aportan sobre todo se refiere a la mantenibilidad del código. Es mucho más fácil entender y mantener un código bien documentado que aquél que no lo está. De esta forma se agiliza el comprendimiento por parte de futuros miembros del proyecto de AGM.

Se ha seguido un estándar determinado de comentarios javadoc a nivel de método.

El estándar es el siguiente:

```
/**
```

```
* @author nombre y apellidos
* @version 1.0 07/2007 (en el caso de que el método se fuera modificando se
iría variando la versión)
*  Descripción de la funcionalidad del método de tal forma que el código del
mismo
* sea fácilmente asimilable
* @param p1 descripción de la funcionalidad del p1 en el ámbito del método
* @return Object descripción de la salida del método
*/
public Object metodo (TipoParametro p1) {… return objeto;}
```

De esta forma, Se lleva cuenta de quienes han sido las personas que han creado o modificado cada método junto con la fecha. Las descripciones tanto de la funcionalidad del mismo método como de los parámetros y el tipo devuelto es esencial para un rápido y eficaz entendimiento.
Si el método anterior fuera modificado por la misma persona se cambiaría la versión del mismo a 1.1 y se actualizaría la fecha y quedaría de esta forma:

```
/**
* @author nombre y apellidos
* @version 1.1 08/2007 (en el caso de que el método se fuera modificando se
iría variando la versión)
*  Descripción de la funcionalidad del método de tal forma que el código del
mismo
* sea fácilmente asimilable
* @param p1 descripción de la funcionalidad del p1 en el ámbito del método
* @return Object descripción de la salida del método
*/
public Object metodo (TipoParametro p1) {… return objeto;}
```

En el caso de que el método haya sido revisado por otro autor, se deben añadir 2 nuevas etiquetas “author” y “version” a continuación de las originales para así llevar cuenta de quién es el responsable de cada versión. Si otro autor hubiera modificado el anterior método quedaría así:

```
/**
* @author nombre y apellidos
* @version 1.1 08/2007 (en el caso de que el método se fuera modificando se
iría variando la versión)
* @author nombre y apellidos
* @version 1.2 08/2007
*  Descripción de la funcionalidad del método de tal forma que el código del
mismo
* sea fácilmente asimilable
* @param p1 descripción de la funcionalidad del p1 en el ámbito del método
* @return Object descripción de la salida del método
*/
public Object metodo (TipoParametro p1) {… return objeto;}
```

El código de AGM está en una buena proporción bien documentado. Los métodos se han ido comentando según se han ido analizando y modificando el código para aplicar buenas prácticas y corregir ciertos bugs. Esto da cuenta de que puede haber métodos que aún estén sin los comentarios javadoc descritos. Esto obligaría a quién se encontrara en esta situación a seguir con la codificación del código AGM y además esta ausencia de comentarios ayuda a identificar código que debería ser repasado.

Hay que tener especial cuidado de no sobreescribir en las clases de los EJB´s los comentarios xdoclet ya que estos son esenciales para la generación de los ficheros EJB. En este caso lo que se debe hacer es comentar el código pero sin tocar estos comentarios.

2.5. Corrección de Bugs

A la vez que se fue modificando el código para adaptarlo a una serie de buenas prácticas y para incluir los comentarios javadoc indicados, se fueron corrigiendo ciertos bugs en el propio desarrollo del videojuego.
La idea fue empezar a corregir bugs en el modo monojugador y para cuándo éste funcionara correctamente, empezar a corregir bugs en el modo multijugador.
Antes de analizar la corrección de bugs de este año, se deberían repasar todos los capítulos relacionados con troubleshooting de los años anteriores para ir llevando cuenta de qué se ha corregido desde un principio.

Cada bug detectado se mostrará en el siguiente formato:

- **Descripción del problema**: con los detalles de cuál es el bug observado.
- **Versión en la que aparece**: cada bug arreglado da cuenta de una nueva versión en la cuál se deberían volver a repasar el resto de bugs que no hubieran sido resueltos ya. La original es la 1.0
- **Pasos para reproducir el problema**: complementa a la descripción para que se tenga total conocimiento del error observado.
- **Explicación del origen del problema**: cuál era la fuente del error
- **Solución aplicada**: descripción de la solución aplicada para llevar cuenta de los cambios.

Primero se explicará cuáles son los pasos que se han seguido para ir resolviendo bugs por si fuera de utilidad en un futuro.
Después se enumerarán los distintos bugs los cuáles algunos están corregidos y otros aún sin corregir.

2.5.1. Consejos para la detección y corrección de bugs

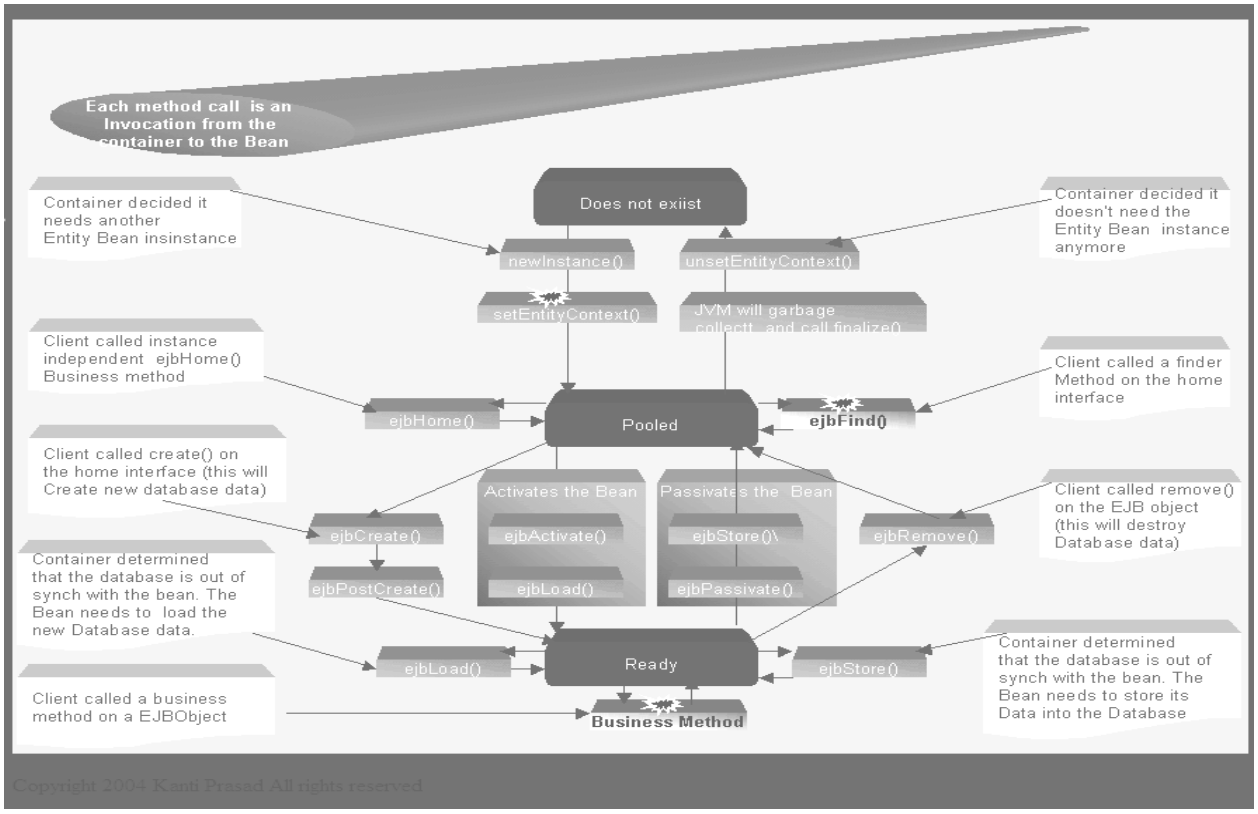
La forma de proceder fue en un principio con el servidor arrancado en modo “Run”. Entonces se fueron observando los fallos que se iban cometiendo. Según se cometía un error, se intentaba encontrar el origen del mismo.
Para encontrar el origen de un problema se puede proceder de 2 formas:

- Ir añadiendo **mensajes por consola** mediante log4j que nos den pistas de cómo avanza la ejecución. Para ello se debería conocer a la perfección cuál es la traza de código por la que se pasa al cometer un error.
- **Depurar el código** mediante la funcionalidad incluida en Eclipse para observar así el valor de las variables de las distintas clases. Dentro de la depuración existen 2 zonas que se prestan a ello:
    - **La parte de código desplegada en el servidor** que se corresponde con los distintos componentes EJB. Para poder depurar aquí es necesario arrancar el servidor en modo “debug”. Bastará con añadir puntos de interrupción en el código del paquete “src” de nuestro proyecto y Lomboz se encargará de

generar la correspondencia entre el código de Jboss y el código del workspace de Eclipse.

- **La parte de código que no se despliega en JBoss** y que se engloba con el código cliente de AGM. Esta parte de código es depurada como es habitual, añadiendo los puntos de interrupción que se deseen y arrancando eso sí ClienteAdministrador o ClienteJugador en modo “debug” de Eclipse.

Al ser un entorno web, pueden llegar a producirse comportamiento erróneos de la aplicación que sean difícilmente reproducibles. Esto hace que ciertos bugs sean muy pesados de corregir. Y sobre todo si el código no se mantiene bien documentado.
Para que la corrección sea lo más rápida posible y así se gane tiempo en el desarrollo de AGM se ha de estar en condiciones suficientes para poder afrontar el proyecto. Ello requiere conocer bien la aplicación y la arquitectura J2EE en la que se basa.
Habiendo entendido todo ello puede llegar a ser muy interesante el ciclo de vida de HabitacionBean que como todo BMP se ajusta al siguiente diagrama obtenido de la siguiente url http://www.comptechdoc.org/docs/kanti/ejb/lifecyclebmp.html



2.7 Ciclo de vida de un BMP

A continuación se enumeran los distintos bugs algunos corregidos y otros por corregir aún ya que se está investigando sobre su origen sin éxito.

2.5.2. Conversación entre jugador y personaje no jugador

- **Descripción del problema:** No finalización de conversaciones entre personaje jugador y personaje no jugador en ciertas ocasiones o pérdidas de ciertos mensajes de la conversación.
- **Versión en la que aparece**: 1.0
- **Pasos para reproducir el problema:** Se conecta un jugador cualquiera a la misión por defecto de AGM. Entonces se inicia una conversación con el Conserje y el botón "Ejecutar" desaparece mientras dura la conversación y tampoco se permite desconectar. Lo que ocurre es que entonces la conversación se queda en un punto en el cual no aparecen opciones de respuesta y no se da por finalizada ya que no vuelven a aparecer ni el botón de "Ejecutar" ni se nos deja cerrar la ventana ya que se dice que se está en medio de una conversación.
  Esto se ha probado en 10 ocasiones y la conversación ha tenido éxito (se ha llegado a concluir) en 3 ocasiones tan sólo.
- **Explicación del origen del problema:** las comunicaciones en AGM entre personajes jugadores y personajes no jugadores se realizan delegando una secuencia de acciones entre el propio ClienteJugador y la habitación en la que se encuentre en cada momento. Una vez se pulsa "Ejecutar" esto después de una serie de pasos desemboca en el envío de un mensaje JMS con el ObjetoComunicacion correspondiente que recibe ClienteJugador.
  Éste lo que hace es delegar las escrituras en pantalla de los distintos mensajes de texto de la conversación a la Interfaz Gráfica (IG). Los ObjetoComunicacion pueden tener una conversación fija y una serie de opciones. Las conversaciones fijas se escriben dando un tiempo de 3 segundos entre ellas. La encargada de escribir es la IG y estos 3 segundos se deben dar para que la interfaz las escriba y el usuario la vea.
  La IG delega toda la escritura de texto a un atributo estático de tipo EscritorTexto que se trata de un hilo. Este objeto se gestiona con el método escribirTexto de IG. El caso es que EscritorTexto también permanece "dormido" durante unos segundos dependiendo de la longitud del mensaje que debe mostrar en cada momento.
  Entonces se observó que algunos mensajes de las conversaciones fijas no daban tiempo a verse así que el fallo estaba en la "pseudosincronización" que se buscó entre la IG y ClienteJugador a la hora de escribir mensajes.
- **Solución aplicada**: Se ha cambiado toda la relación que existía antiguamente entre que se recibe el mensaje JMS en ClienteJugador hasta que se escribe por pantalla.
  Ahora ClienteJugador no espera 3 segundos entre cada frase de la conversación fija sino que los lanza todos en una colección. Cada fragmento de la conversación se encapsula en un FragmentoConversacion que lleva la misma información que anteriormente. Entonces la IG lo que hace ahora es recorrer esta lista con los FragmentoConversacion y lanzar un hilo de distinta prioridad según el orden en que marca la lista. Cada hilo (HiloComunicacion) tiene un objeto EscritorTexto que es exactamente el mismo que el de la versión anterior de AGM. Lo que pasa es que ahora cada uno llama a un método synchronized "escribirTexto" que junto con un entero estático de la clase obliga a mantener el orden en los mensajes y es este método el que ahora sí deja el hilo dormido unos segundos dependiendo del mensaje a mostrar en cada momento.
  Se han probado 10 conversaciones con el Conserje y no se pierde ningún mensaje en ellas y todas acaban exitosamente. Se continúa con la **versión 1.1**

2.5.3. Cierre de las sesiones JMS

- **Descripción del problema**: En determinadas ocasiones en la consola del JBoss aparece la siguiente traza:

```
17:53:10,500 WARN  [OILServerILService] Connection failure (1).
java.net.SocketException: Connection reset
        at java.net.SocketInputStream.read(SocketInputStream.java:168)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:183)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:201)
        at java.io.ObjectInputStream$PeekInputStream.peek(ObjectInputStream.java:2123)
        at
java.io.ObjectInputStream$BlockDataInputStream.readBlockHeader(ObjectInputStream.java:23
03)
        at
java.io.ObjectInputStream$BlockDataInputStream.refill(ObjectInputStream.java:2370)
        at
java.io.ObjectInputStream$BlockDataInputStream.read(ObjectInputStream.java:2442)
        at
java.io.ObjectInputStream$BlockDataInputStream.readByte(ObjectInputStream.java:2591)
        at java.io.ObjectInputStream.readByte(ObjectInputStream.java:845)
        at org.jboss.mq.il.oil.OILServerILService$Client.run(OILServerILService.java:205)
        at java.lang.Thread.run(Thread.java:534)
17:53:10,500 WARN  [OILServerILService] Connection failure (1).
java.net.SocketException: Connection reset
        at java.net.SocketInputStream.read(SocketInputStream.java:168)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:183)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:201)
        at java.io.ObjectInputStream$PeekInputStream.peek(ObjectInputStream.java:2123)
        at
java.io.ObjectInputStream$BlockDataInputStream.readBlockHeader(ObjectInputStream.java:23
03)
        at
java.io.ObjectInputStream$BlockDataInputStream.refill(ObjectInputStream.java:2370)
        at
java.io.ObjectInputStream$BlockDataInputStream.read(ObjectInputStream.java:2442)
        at
java.io.ObjectInputStream$BlockDataInputStream.readByte(ObjectInputStream.java:2591)
        at java.io.ObjectInputStream.readByte(ObjectInputStream.java:845)
        at org.jboss.mq.il.oil.OILServerILService$Client.run(OILServerILService.java:205)
        at java.lang.Thread.run(Thread.java:534)
```

Siempre aparece por ejemplo justo después de crear un jugador.

- **Versión en la que aparece**: Versión 1.1
- **Pasos para reproducir el problema:** Creamos un jugador y observamos la consola. Además una vez conectados a AGM, cuando nos desconectamos vuelve a salir
- **Explicación del origen del problema**: se debe a que no se cierran la QueueConnection y la TopicConnection cuando se acaba de alguna forma la ejecución de ClienteJugador.
- **Solución aplicada**: se ha creado un método en ClienteJugador, *"cierraClienteJugador"* que sustituye a las antiguas llamadas a System.exit. El método lo que hace es cerrar las conexiones JMS antes de llamar a System.exit. Se libera entonces la **versión 1.2** de AGM.

2.5.4. Excepción al tiempo de inactividad del servidor

- **Descripción del problema:** una vez arrancamos el servidor con el despliegue del módulo de EJB de AGM, si no realizamos ninguna acción durante un breve espacio de tiempo se obtiene la siguiente excepción:

```
19:40:24,109 WARN  [AbstractInstanceCache] failed to passivate, id=f5ttelqt-e
javax.ejb.EJBException: Could not passivate; failed to save state; CausedByException is:
        org.apache.log4j.Logger
        at
org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager.passivateSession(StatefulSes
sionFilePersistenceManager.java:378)
        at
org.jboss.ejb.plugins.StatefulSessionInstanceCache.passivate(StatefulSessionInstanceCach
e.java:85)
        at
org.jboss.ejb.plugins.AbstractInstanceCache.tryToPassivate(AbstractInstanceCache.java:15
6)
        at
org.jboss.ejb.plugins.AbstractInstanceCache.release(AbstractInstanceCache.java:192)
        at
org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy.ageOut(LRUEnterpriseContextCachePo
licy.java:274)
        at
org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy$OveragerTask.kickOut(LRUEnterprise
ContextCachePolicy.java:446)
        at
org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy$OveragerTask.run(LRUEnterpriseCont
extCachePolicy.java:405)
        at java.util.TimerThread.mainLoop(Timer.java:432)
        at java.util.TimerThread.run(Timer.java:382)
java.io.NotSerializableException: org.apache.log4j.Logger
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1059)
        at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1337)
        at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1309)
        at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1252)
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1057)
        at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:278)
        at
org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager.passivateSession(StatefulSes
sionFilePersistenceManager.java:370)
        at
org.jboss.ejb.plugins.StatefulSessionInstanceCache.passivate(StatefulSessionInstanceCach
e.java:85)
        at
org.jboss.ejb.plugins.AbstractInstanceCache.tryToPassivate(AbstractInstanceCache.java:15
6)
        at
org.jboss.ejb.plugins.AbstractInstanceCache.release(AbstractInstanceCache.java:192)
        at
org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy.ageOut(LRUEnterpriseContextCachePo
licy.java:274)
        at
org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy$OveragerTask.kickOut(LRUEnterprise
ContextCachePolicy.java:446)
        at
org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy$OveragerTask.run(LRUEnterpriseCont
extCachePolicy.java:405)
        at java.util.TimerThread.mainLoop(Timer.java:432)
at java.util.TimerThread.run(Timer.java:382)
```

- **Versión en la que aparece**: versión 1.2
- **Pasos para reproducir el problema**: tan solo mantenemos el servidor arrancado y periódicamente se muestra la excepción.
- **Explicación del origen del problema**: Cada EJB tiene un método ejbPassivate. Este método lo invoca el contenedor cuando en el mismo ya existen muchas instancias de EJB y necesita pasar a “segundo plano”. El contenedor lo que hace es escribir el objeto en un almacén temporal y para ello los campos y la misma

clase deben ser serializables.
El problema es que las variables de tipo Logger no lo son por lo que no pueden "pasar a segundo plano". Y no lo deberían ser ya que no contribuyen a formar el estado del objeto que es lo que se debe almacenar para que luego más tarde el contenedor vuelva a requerir este bean.

- **Solución aplicada**: Los atributos de las clases Bean de AGM de tipo Logger se han hecho transient para que así no se tengan en cuenta a la hora de serializar objetos de esas clases.
  Se continúa entonces con la **versión 1.3**

2.5.5. Excepción JMS

- Descripción del problema: Al conectarse con cierto jugador y realizar ciertas operaciones aparece un error en consola con la siguiente traza

```
[2007/08/29 20:10:06.328] org.jboss.mq.SpyMessageConsumer Message consumer closing due
to error in listening thread.
org.jboss.mq.SpyJMSException: Cannot create a ConnectionReceiver; - nested throwable:
(java.lang.NullPointerException)
        at org.jboss.mq.Connection.receive(Connection.java:1178)
        at org.jboss.mq.SpyMessageConsumer.run(SpyMessageConsumer.java:487)
        at java.lang.Thread.run(Thread.java:534)
Caused by: java.lang.NullPointerException
        at org.jboss.mq.server.JMSTopic.receive(JMSTopic.java:260)
        at org.jboss.mq.server.ClientConsumer.receive(ClientConsumer.java:225)
        at
org.jboss.mq.server.JMSDestinationManager.receive(JMSDestinationManager.java:672)
        at
org.jboss.mq.server.JMSServerInterceptorSupport.receive(JMSServerInterceptorSupport.java
:225)
        at
org.jboss.mq.security.ServerSecurityInterceptor.receive(ServerSecurityInterceptor.java:1
03)
        at org.jboss.mq.server.TracingInterceptor.receive(TracingInterceptor.java:478)
        at org.jboss.mq.server.JMSServerInvoker.receive(JMSServerInvoker.java:227)
        at org.jboss.mq.il.oil.OILServerILService$Client.run(OILServerILService.java:294)
        ... 1 more
```

Esta excepción se repite cada vez que se cambia de habitación o se realiza alguna acción en ella. También cuando se desconectan jugadores del juego.
Pero curiosamente esto no ocurría justo al conectarse.

- **Versión en la que aparece**: Versión 1.3

- **Pasos para reproducir el problema**: Se conecta un jugador como de costumbre y se realizan operaciones como cambiar de habitación, hablar con el Conserje, etc…

- **Explicación del origen del problema**: Se buscó por Internet en numerosos foros dónde la gente preguntaba por excepciones similares en distintas versiones de Jboss sin ninguna respuesta que tuviera una explicación clara y concisa sobre qué estaba ocurriendo y cómo solucionarlo.
  Entonces se decidió seguir la traza e investigar por cuenta ajena.
  Analizando el código se persiguió con especial atención que todas las conexiónes JMS (tanto TopicConnection y QueueConnection) de ClienteJugador se gestionaran adecuadamente.
  Se observó entonces el método subscribeA que detallo a continuación

```
/**
         * @author Juan Carlos Castromil
         * @version 1.0 07/2007
         * @param topicString
         * @param messageSelector
         * @param ml
         * @return el TopicSubscriber
         */
        public TopicSubscriber subscribeA(String topicString,
                        String messageSelector, MessageListener ml) {

                Topic topic;
                try {

                        topic = (javax.jms.Topic) ctx.lookup(topicString);
                        TopicSession session = tC.createTopicSession(true, 1);
                        java.net.InetAddress i = java.net.InetAddress.getLocalHost();

                        subscriptor = session.createDurableSubscriber(topic,
                                        i.getHostAddress(), messageSelector, true);

                        subscriptor.setMessageListener(ml);
                        tC.start();
                } catch (NamingException e) {
                        JOptionPane
                        .showMessageDialog(null,
                        "Primero se debe ejecutar la aplicación ClienteAdministrador");
                        cierraClienteJugador(0);
                } catch (JMSException e) {
                        log.error("Error JMS en subscribeA " + e.getMessage());
                        e.printStackTrace();
                } catch (UnknownHostException e) {
                        log.error("Host Desconocido " + e.getMessage());
                        e.printStackTrace();
                }

                return subscriptor;
        }
```

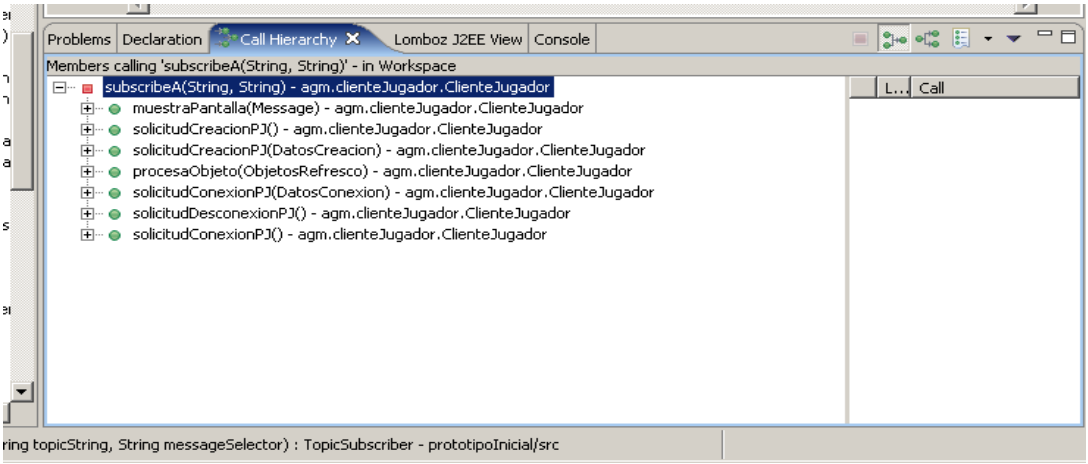
El caso es que cada vez que se invoca este método se invoca “start” sobre el atributo de ClienteJugador TopicConnection y esto se hacía cada vez que se invocaba incluso cuando antes no se había creado ni tan siquiera ninguna conexión nueva. Incluso cuando si se había creado, se hacía start sobre la nueva conexión pero la anterior entonces quedaba abierta.

- **Solución aplicada:** Se optó entonces porque cada vez que se llamara al método se comprobara que si la TopicConnection no era nula y si así ocurría se cerrara y se creara una nueva. El método entonces quedó así y quedaba resuelta la incidencia**:**

```
/**
         * @author Juan Carlos Castromil
         * @version 1.0 07/2007
         * MODIFICACION :Se cierra la anterior
         * TopicConnection si existía y
         * y se crea una nueva antes de hacer el resto de la
         * funcionalidad del método
         * @param topicString es el nombre de la cola a buscar
         * @param messageSelector es el selector del mensaje
         * @return el TopicSubscriber
         */
        private TopicSubscriber subscribeA(String topicString,
                        String messageSelector){
                return subscribeA(topicString, messageSelector, this);
        }

        /**
         * @author Juan Carlos Castromil
```

```
         * @version 1.0 08/2007
         * @param topicString es el id del topic al que
         * se va a suscribir ClienteJugador
         * @param messageSelector
         * @param ml
         * @return el TopicSubscriber
         */
        public TopicSubscriber subscribeA(String topicString,
                        String messageSelector, MessageListener ml) {

                Topic topic;
                try {
                        //cerramos la anterior conexión en el que caso de que existiera
                        if (tC != null) {
                                tC.close();
                        }
                        tC = creaTopicConnection();
                        topic = (javax.jms.Topic) ctx.lookup(topicString);
                        TopicSession session = tC.createTopicSession(true, 1);
                        java.net.InetAddress i = java.net.InetAddress.getLocalHost();

                        subscriptor = session.createDurableSubscriber(topic,
                                        i.getHostAddress(), messageSelector, true);

                        subscriptor.setMessageListener(ml);
                        tC.start();
                } catch (NamingException e) {
                        JOptionPane
                        .showMessageDialog(null,
                        "Primero se debe ejecutar la aplicación ClienteAdministrador");
                        cierraClienteJugador(0);
                } catch (JMSException e) {
                        log.error("Error JMS en subscribeA " + e.getMessage());
                        e.printStackTrace();
                } catch (UnknownHostException e) {
                        log.error("Host Desconocido " + e.getMessage());
                        e.printStackTrace();
                }

                return subscriptor;
        }
```

Este método se invoca desde numerosos métodos de ClienteJugador. La “Call Hierarchy” es la siguiente:



2.7 Call Hierarchy de “subscribeA”

Se explica así porque se producía anteriormente en tantas ocasiones la excepción indicada al principio.
Se continúa entonces con **la versión 1.4**

2.5.6. Mala programación de ProfesorSOS.java

- **Descripción del problema**: al realizar la acción “USAR” la “Llave_Taquilla” sobre el “ProfesorSOS” no aparece nada.
- **Versión en la que aparece**: versión 1.4
- **Pasos para reproducir el problema**: tan solo se trata de realizar la acción antes comentada.
- **Explicación del origen del problema:** al realizarse la acción se producía por consola de Jboss la siguiente excepción:

```
9:16:44,112 ERROR [STDERR] java.lang.NullPointerException
19:16:44,112 ERROR [STDERR]
at
agm.servidor.habitaciones.HabitacionBean.ejecutaAccion(HabitacionBean.java:558)
```

Esto se corresponde con la siguiente línea de HabitacionBean

```
    //se da por hecho que se encontrará entre ellos entonces..
   //Según el personajeNoJugador se ejecutara un ejecutarAccion
   //distinto
   Acciones acciones = (Acciones) p.ejecutarAccion(idAccion, idO2,
   idActor);
   //el ejecutar de cada accion se han definido en cada personajeNoJugador
   acciones.ejecutar(idHab, idActor);
```

Y esto es debido a que las acciones devueltas por p (en este caso ProfesorSOS) son nulas. Si se observaba el código ejecutaAccion de ProfesorSOS.java se observaba el siguiente trozo del mismo:

```
…
} else if ("USAR".equals(idA)) {
                Habitacion h = Acciones.conseguirHabitacion(idHab);
                String nombre;
                ObjetoNoPersonajeClase oNPC = null;
                if (idO != null) {
                    oNPC = h.buscarObjeto(idO, idPj);
                    if (oNPC != null) {
                        String tipo = oNPC.getTipo();
                        int ind = tipo.lastIndexOf(".");
                        nombre = tipo.substring(ind + 1);
                    } else
                        nombre = "";
                } else
                    nombre = "";
                Acciones a = null;
                if ((nombre.equals("CarneJ")) && (estado.equals("ESPERANDO"))) {
                    a = teRespondo("OPCION 2.7", idPj);
                }
                if ((nombre.equals("CarneI")) && (estado.equals("ESPERANDO"))) {
                    PJs.add(idPj);
                    guardarLista();
                    a = teRespondo("OPCION 2.7b", idPj);
                }
                return a;
            } else {…
```

Entonces lo que está ocurriendo es que no se contempla la acción devuelta en el caso de que se ejecute "USAR" sobre ProfesorSOS y el objeto no sea ni un carné ni otro. Por lo tanto hay que rellenar esto.

- **Solución aplicada**: se modifica el método "ejecutarAccion" de ProfesorSOS.java para que en el caso de que se realice sobre él la acción USAR con algo distinto a los carnés I o J se devuelva una acción distinta de null.
  Se lanza así la **versión 1.5**

2.5.7. Actualización de objetos en inventario

- **Descripción del problema**: En las conversaciones entre personaje y personaje no jugador que surgen como consecuencia de una acción "USAR" en la que además se actualice un objeto en el inventario se pierden ciertos mensajes y además el objeto que se actualiza aparece antes que la propia conversación

- **Versión en la que aparece**: Versión 1.5

- **Pasos para reproducir el problema**: Se inicia la misión por defecto y se obtiene el CarnéI en Secretaría. Después se dirigen al despacho del ProfesorSOS y se le entrega dicho carné y se produce el "bug".

- **Explicación del origen del problema:** la acción programada en ProfesorSOS.java cuando se usa el CarnéI sobre él acaba desembocando en 2 métodos que corren en paralelo:

  - **mostrarResultado:** este lo que hace es procesar una conversación entre personaje no jugador y jugador que "debería" ser antecesora de la actualización del objeto en el inventario. Acaba creando una variable local "EscritorTexto" que se encarga de la conversación.
  - **meterEnInventario:** este método lo que hace es modificar el personaje jugador para que en sus objetos aparezca este nuevo objeto mandar. Al final desde HabitacionBean se acaba mandando un mensaje JMS al ClienteJugador correspondiente con un objetoActualización para actualizar la IG. Esta actualización conlleva también la escritura de texto en pantalla al igual que se hace con la conversación entre el jugador y el no jugador. Pero esta escritura de texto se hace con un objeto EscritorTexto que es global a IG. Es decir, ambos métodos terminan usando distintas instancias de EscritorTexto pero ambos escriben sobre el mismo areaTexto.

Lo que ocurre es que se está invocando métodos distintos que acaban escribiendo en la interfaz gráfica y el segundo método no espera al primero por lo que se solapan las acciones sobre la Interfaz Gráfica. Se solucionaría por lo tanto el problema si de alguna forma mantenemos la sincronización entre "mostrarResultado" y "meterEnInventario" de tal forma que hasta que el primero no acabe (no acaben todos los hilos que lanza) no se ejecute el segundo. El código que conlleva este bug es el siguiente:

```
…
mostrarResultado(h, o);
```

```
meterEnInventario(h, idPJ,
                                    "agm.objetos.objetosNoPersonaje.AprobadoSOS",
                                    "¡Aprobe!");
…
```

Entonces entre ambos hay que ingeniárselas para que se sincronicen. Una solución “fácil” es añadir un delay entre ambos métodos de tal forma que se de tiempo a que el hilo lanzado desde mostrarResultado finalice pero esto no es ni mucho menos la solución más elegante ya que para que funcione bien deberíamos dar un tiempo mayor o igual al que necesite mostrarResultado e incluso este tiempo puede llegar a ser indeterminado dentro de un rango fijo.

Ambos métodos desembocan en otros 2 métodos de InterfazGrafica que son escribirConversación (mostrarResultado) y escribirTexto (meterEnInventario). La sincronización la vamos a hacer mediante un tipo de objeto que comparten estos 2 métodos. Este tipo es EscritorTexto.

Como solución al bug 2.5.2 se creó el método escribirConversación al que hacemos referencia y se prefirió para solucionar ese problema, que en el método se creara una variable local EscritorTexto distinta de otra global del mismo tipo que había ya en IG. Pero para esta sincronización será necesario que usen la misma así que se modifica el código que se añadió por aquel entonces para que no se use una variable local EscritorTexto sino que se utilice la misma que la global.

Entonces la sincronización la conseguimos mediante un atributo estático booleano añadido en EscritorTexto de tal forma que cuando se ejecuta el método *“public synchronized void escribeTexto(FragmentoConversacion fragmento, ClienteJugador cJ, int contadorL)”* (el que se lanza cuando se está procesando una conversación entre personaje jugador y no jugador) este atributo se pone a true y cuando se ha escrito la última línea de la conversación se pone a false.

Entonces desde escribirTexto se pregunta por el valor de esta variable que hasta que no sea false se duerme en intervalos de medio segundo el hilo propietario de la llamada. Una vez es false, se pasa a escribir el texto que corresponda.

De esta forma nunca se escribirá texto de una conversación entre personaje jugador y no jugador y texto de otra conversación cualquiera a la vez sino que se turnarán.

Se continúa entonces con la **versión 1.6**

2.5.8. Mal cambio de habitación

- **Descripción del problema**: Al cambiar de habitación en muy rara ocasión el jugador no aparece en la nueva habitación y no se puede ejecutar nada sobre la nueva habitación.
- **Versión en la que aparece**: Versión 1.6
- **Pasos para reproducir el problema**: Al darse en muy extrañas ocasiones, los pasos para reproducir el problema son indescriptibles.
- **Explicación del origen del problema**: Cuando esto ocurre en la consola del Jboss sale lo siguiente:

```
21:36:17,531 INFO  [HabitacionBMP] Se va a ejecutar la acción IR A en la habitacion H13
con el idO1 O29 e idO2
21:36:17,531 INFO  [HabitacionBMP] Se va a buscar el objeto O29 en el inventario
21:36:17,531 INFO  [HabitacionBMP] No se ha encontrado el objeto de id O29en el
personaje p
21:36:17,531 INFO  [HabitacionBMP] Se va a buscar el objeto de O29en la habitacion H13
```

```
21:36:17,750 INFO  [HabitacionBMP] Se va a proceder a sacar al personaje de id p de la
habitacion H13
21:36:17,750 ERROR [LogInterceptor] TransactionRolledbackException, causedBy:
java.lang.NullPointerException
        at
agm.servidor.habitaciones.HabitacionBean.sacaPersonajeJ(HabitacionBean.java:785)
        at sun.reflect.GeneratedMethodAccessor102.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.ejb.EntityContainer$ContainerInterceptor.invoke(EntityContainer.java:1044)
        at
org.jboss.ejb.plugins.EntitySynchronizationInterceptor.invoke(EntitySynchronizationInter
ceptor.java:301)
        at
org.jboss.resource.connectionmanager.CachedConnectionInterceptor.invoke(CachedConnection
Interceptor.java:186)
        at
org.jboss.ejb.plugins.EntityReentranceInterceptor.invoke(EntityReentranceInterceptor.jav
a:82)
        at
org.jboss.ejb.plugins.EntityInstanceInterceptor.invoke(EntityInstanceInterceptor.java:17
4)
        at
org.jboss.ejb.plugins.EntityLockInterceptor.invoke(EntityLockInterceptor.java:89)
        at
org.jboss.ejb.plugins.EntityCreationInterceptor.invoke(EntityCreationInterceptor.java:53
)
        at
org.jboss.ejb.plugins.AbstractTxInterceptor.invokeNext(AbstractTxInterceptor.java:84)
        at
org.jboss.ejb.plugins.TxInterceptorCMT.runWithTransactions(TxInterceptorCMT.java:243)
        at org.jboss.ejb.plugins.TxInterceptorCMT.invoke(TxInterceptorCMT.java:104)
        at org.jboss.ejb.plugins.SecurityInterceptor.invoke(SecurityInterceptor.java:117)
        at org.jboss.ejb.plugins.LogInterceptor.invoke(LogInterceptor.java:191)
        at
org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor.invoke(ProxyFactoryFinderInterceptor
.java:122)
        at org.jboss.ejb.EntityContainer.internalInvoke(EntityContainer.java:483)
        at org.jboss.ejb.Container.invoke(Container.java:674)
        at sun.reflect.GeneratedMethodAccessor58.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:28
4)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:549)
        at org.jboss.invocation.local.LocalInvoker.invoke(LocalInvoker.java:101)
        at org.jboss.invocation.InvokerInterceptor.invoke(InvokerInterceptor.java:83)
        at org.jboss.proxy.TransactionInterceptor.invoke(TransactionInterceptor.java:46)
        at org.jboss.proxy.SecurityInterceptor.invoke(SecurityInterceptor.java:45)
        at org.jboss.proxy.ejb.EntityInterceptor.invoke(EntityInterceptor.java:97)
        at org.jboss.proxy.ClientContainer.invoke(ClientContainer.java:85)
        at $Proxy67.sacaPersonajeJ(Unknown Source)
        at agm.objetos.objetosNoPersonaje.Acciones.cambioHabitacion(Acciones.java:87)
        at agm.objetos.objetosNoPersonaje.Puerta$2.ejecutar(Puerta.java:108)
        at
agm.servidor.habitaciones.HabitacionBean.ejecutaAccion(HabitacionBean.java:562)
        at sun.reflect.GeneratedMethodAccessor107.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.ejb.EntityContainer$ContainerInterceptor.invoke(EntityContainer.java:1044)
        at
org.jboss.ejb.plugins.EntitySynchronizationInterceptor.invoke(EntitySynchronizationInter
ceptor.java:301)
        at
org.jboss.resource.connectionmanager.CachedConnectionInterceptor.invoke(CachedConnection
Interceptor.java:186)
        at
org.jboss.ejb.plugins.EntityReentranceInterceptor.invoke(EntityReentranceInterceptor.jav
a:82)
```

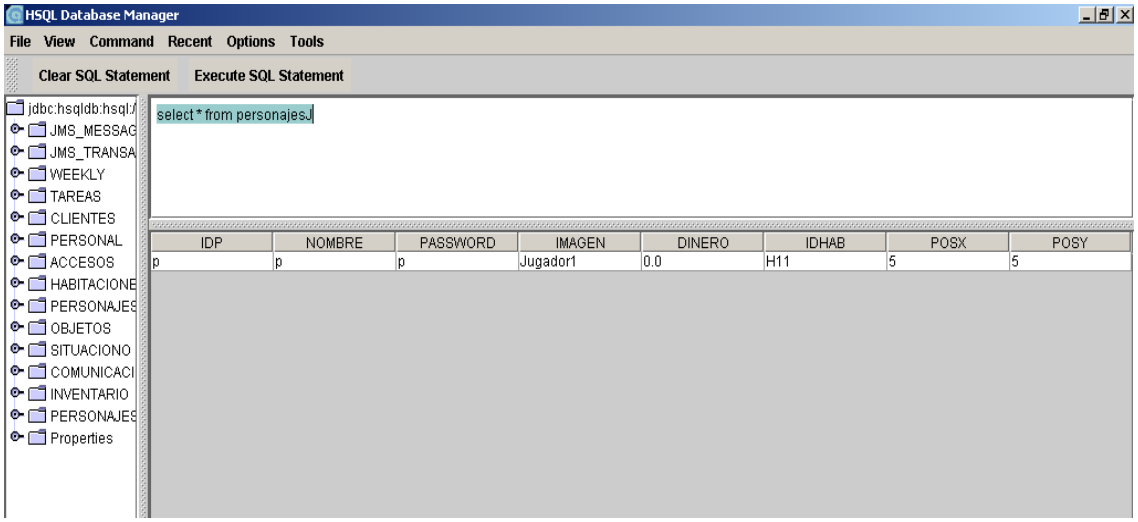
```
        at
org.jboss.ejb.plugins.EntityInstanceInterceptor.invoke(EntityInstanceInterceptor.java:17
4)
        at
org.jboss.ejb.plugins.EntityLockInterceptor.invoke(EntityLockInterceptor.java:89)
        at
org.jboss.ejb.plugins.EntityCreationInterceptor.invoke(EntityCreationInterceptor.java:53
)
        at
org.jboss.ejb.plugins.AbstractTxInterceptor.invokeNext(AbstractTxInterceptor.java:84)
        at
org.jboss.ejb.plugins.TxInterceptorCMT.runWithTransactions(TxInterceptorCMT.java:243)
        at org.jboss.ejb.plugins.TxInterceptorCMT.invoke(TxInterceptorCMT.java:104)
        at org.jboss.ejb.plugins.SecurityInterceptor.invoke(SecurityInterceptor.java:117)
        at org.jboss.ejb.plugins.LogInterceptor.invoke(LogInterceptor.java:191)
        at
org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor.invoke(ProxyFactoryFinderInterceptor
.java:122)
        at org.jboss.ejb.EntityContainer.internalInvoke(EntityContainer.java:483)
        at org.jboss.ejb.Container.invoke(Container.java:674)
        at sun.reflect.GeneratedMethodAccessor58.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:28
4)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:549)
        at org.jboss.invocation.jrmp.server.JRMPInvoker.invoke(JRMPInvoker.java:359)
        at sun.reflect.GeneratedMethodAccessor92.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:261)
        at sun.rmi.transport.Transport$1.run(Transport.java:148)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
        at
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
        at java.lang.Thread.run(Thread.java:534)
```

La cima de la pila se corresponde con este código:

```
/**
     * @ejb.interface-method
     * @author Juan Carlos Castromil
     * @version 1.0 07/2007
     * @param idPJ
     * @param textoAMostrar
     * @param textoMostrarAlResto
     * @return el personaje
     */
    public PersonajeJugador sacaPersonajeJ(String idPJ, String textoAMostrar,
            String textoAMostrarAlResto) {
        log.info("Se va a proceder a sacar al personaje de id " + idPJ + " de la
habitacion " + this.idHab);
        //este método presupone que el jugador en cuestión será encontrado
        //mala práctica en un entorno web
        PersonajeJugador p = buscarPersonaje(idPJ);
        //si la colección cambia se devuelve true después de llamar a remove
         boolean b = personajesJ.remove(p);
         //cambiamos la colección personajesJ de la habitacion
         setPersonajesJ(personajesJ);
         //creamos un objeto para actualizar el cambio
         ObjetoActualizacion oA = new ObjetoActualizacion();
         //los pasos siguientes son crear una lista con acciones
         //que serán notificadas al resto de jugadores
         LinkedList lista = new LinkedList();
```

```
        //se añade la acción
        lista.add(new SacaPJ(textoAMostrar, textoAMostrarAlResto, idPJ, p
                .getNombre()));
```

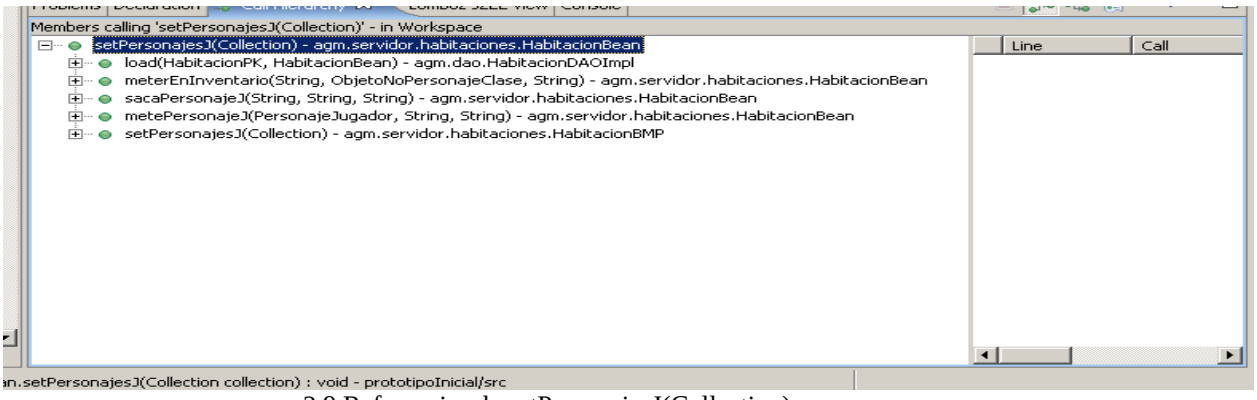
De aquí se deduce que no se ha encontrado al jugador en el método "buscarPersonaje".
Buscamos en la BD y realmente es que el jugador ya se ha movido de alguna forma y por lo tanto no se podía ir a buscar.



2.8 HSQL DataBase Manager

Entonces ¿cuál ha podido ser el problema?
¿Se ha movido antes de alguna forma y se intenta mover ahora de nuevo?

El BMP se actualiza con store cada vez que se modifican algunos de sus atributos para así guardarlos en BD. De alguna forma se ha actualizado la colección de personajes jugadores de la habitación origen antes de tiempo. Y esto hay que pensar que solo ocurre cada mucho tiempo y no es lo más corriente. Una forma podría ser analizar los puntos en los cuales la colección de personajes jugadores de cierta habitación se modifican y eso es seguir las trazas referentes a setPersonajesJ de HabitacionBean. Esta es la siguiente:



2.9 Referencias de setPersonajesJ(Collection)

Se optó entonces por añadir el siguiente trozo de código en el método sacaPersonajeJ para añadir un punto de interrupción en el mismo y así poder depurar en el punto en el que se volviera a reproducir el problema para poder sacar más datos que nos den pistas sobre el origen del problema.

```
//este método presupone que el jugador en cuestión será encontrado
        //mala práctica en un entorno web
        PersonajeJugador p = buscarPersonaje(idPJ);
        //prueba para un bug
        if (p == null) {
                log.error("Esto es síntoma de un error");
        }
```

- **Solución aplicada**: ...

2.5.9. Mensaje inesperado

- **Descripción del problema:** En determinadas acciones del juego, al ejecutar un "Ir a" se muestra un mensaje JOptionPane indicando que "*Primero se debe ejecutar la aplicación ClienteAdministrador*"

- **Versión en la que aparece:** Version 1.6

- **Pasos para reproducir el problema:** Al darse en muy extrañas ocasiones, los pasos para reproducir el problema son difícilmente descriptibles y reproducibles.
  Una vez se conecta cierto jugador, se empieza a tan solo cambiar de habitación continuamente en busca de este error.
  Al cabo de un tiempo, aparece el mensaje por pantalla y en la consola de ClienteJugador aparece la siguiente excepción:

```
java.lang.NullPointerException
        at
agm.dao.HabitacionDAOImpl.actualizaAtributosSimplesDeHabitacion(HabitacionDAOImpl.java:8
64)
        at agm.dao.HabitacionDAOImpl.store(HabitacionDAOImpl.java:902)
        at agm.servidor.habitaciones.HabitacionBMP.ejbStore(HabitacionBMP.java:222)
        at sun.reflect.GeneratedMethodAccessor71.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.ejb.plugins.BMPPersistenceManager.storeEntity(BMPPersistenceManager.java:447)
        at
org.jboss.resource.connectionmanager.CachedConnectionInterceptor.storeEntity(CachedConne
ctionInterceptor.java:388)
        at org.jboss.ejb.EntityContainer.storeEntity(EntityContainer.java:702)
        at
org.jboss.ejb.GlobalTxEntityMap.synchronizeEntities(GlobalTxEntityMap.java:163)
        at
org.jboss.ejb.GlobalTxEntityMap$GlobalTxEntityMapCleanup.beforeCompletion(GlobalTxEntity
Map.java:227)
        at org.jboss.tm.TransactionImpl.doBeforeCompletion(TransactionImpl.java:1297)
        at org.jboss.tm.TransactionImpl.commit(TransactionImpl.java:338)
        at
org.jboss.ejb.plugins.TxInterceptorCMT.endTransaction(TxInterceptorCMT.java:369)
        at
org.jboss.ejb.plugins.TxInterceptorCMT.runWithTransactions(TxInterceptorCMT.java:253)
        at org.jboss.ejb.plugins.TxInterceptorCMT.invoke(TxInterceptorCMT.java:104)
```

```
        at org.jboss.ejb.plugins.SecurityInterceptor.invoke(SecurityInterceptor.java:117)
        at org.jboss.ejb.plugins.LogInterceptor.invoke(LogInterceptor.java:191)
        at
org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor.invoke(ProxyFactoryFinderInterceptor
.java:122)
        at org.jboss.ejb.EntityContainer.internalInvoke(EntityContainer.java:483)
        at org.jboss.ejb.Container.invoke(Container.java:674)
        at sun.reflect.GeneratedMethodAccessor57.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:28
4)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:549)
        at org.jboss.invocation.jrmp.server.JRMPInvoker.invoke(JRMPInvoker.java:359)
        at sun.reflect.GeneratedMethodAccessor90.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:261)
        at sun.rmi.transport.Transport$1.run(Transport.java:148)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
        at
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
        at java.lang.Thread.run(Thread.java:534)
```

Y a la vez tb aparecen los siguientes mensajes en la consola del Jboss:

```
19:31:56,406 INFO  [HabitacionBMP] Se va a ejecutar la acción IR A en la habitacion H10
con el idO1 O24 e idO2
19:31:56,421 INFO  [HabitacionBMP] Se va a buscar el objeto O24 en el inventario
19:31:56,421 INFO  [HabitacionBMP] El jugador p pertenece a la habitacion H10
19:31:56,421 INFO  [HabitacionBMP] No se ha encontrado el objeto de id O24en el
personaje p
19:31:56,421 INFO  [HabitacionBMP] Se va a buscar el objeto de O24en la habitacion H10
19:31:56,453 INFO  [HabitacionBMP] Se va a proceder a sacar al personaje de id p de la
habitacion H10
19:31:56,453 INFO  [HabitacionBMP] Se acaba de publicar un mensaje JMS con
notificaciones textuales de un objetoActualizacion
19:31:56,453 INFO  [HabitacionBMP] Se ha sacado correctamente al jugador de la
habitación
19:31:56,546 INFO  [HabitacionBMP] Se acaba de publicar un mensaje JMS con un objeto
refresco en la habitacionnull
19:31:58,562 WARN  [SecurityManager] No SecurityMetadadata was available for H11 adding
default security conf
19:31:58,562 INFO  [HabitacionBMP] Se acaba de publicar un mensaje JMS con
notificaciones textuales de un objetoActualizacion
19:31:58,578 ERROR [LogInterceptor] TransactionRolledbackException, causedBy:
java.lang.NullPointerException
        at
agm.dao.HabitacionDAOImpl.actualizaAtributosSimplesDeHabitacion(HabitacionDAOImpl.java:8
64)
        at agm.dao.HabitacionDAOImpl.store(HabitacionDAOImpl.java:902)
        at agm.servidor.habitaciones.HabitacionBMP.ejbStore(HabitacionBMP.java:222)
        at sun.reflect.GeneratedMethodAccessor66.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.ejb.plugins.BMPPersistenceManager.storeEntity(BMPPersistenceManager.java:447)
        at
org.jboss.resource.connectionmanager.CachedConnectionInterceptor.storeEntity(CachedConne
ctionInterceptor.java:388)
        at org.jboss.ejb.EntityContainer.storeEntity(EntityContainer.java:702)
        at
org.jboss.ejb.GlobalTxEntityMap.synchronizeEntities(GlobalTxEntityMap.java:163)
        at
org.jboss.ejb.GlobalTxEntityMap$GlobalTxEntityMapCleanup.beforeCompletion(GlobalTxEntity
Map.java:227)
        at org.jboss.tm.TransactionImpl.doBeforeCompletion(TransactionImpl.java:1297)
```

```
        at org.jboss.tm.TransactionImpl.commit(TransactionImpl.java:338)
        at
org.jboss.ejb.plugins.TxInterceptorCMT.endTransaction(TxInterceptorCMT.java:369)
        at
org.jboss.ejb.plugins.TxInterceptorCMT.runWithTransactions(TxInterceptorCMT.java:253)
        at org.jboss.ejb.plugins.TxInterceptorCMT.invoke(TxInterceptorCMT.java:104)
        at org.jboss.ejb.plugins.SecurityInterceptor.invoke(SecurityInterceptor.java:117)
        at org.jboss.ejb.plugins.LogInterceptor.invoke(LogInterceptor.java:191)
        at
org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor.invoke(ProxyFactoryFinderInterceptor
.java:122)
        at org.jboss.ejb.EntityContainer.internalInvoke(EntityContainer.java:483)
        at org.jboss.ejb.Container.invoke(Container.java:674)
        at sun.reflect.GeneratedMethodAccessor52.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at
org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:28
4)
        at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:549)
        at org.jboss.invocation.jrmp.server.JRMPInvoker.invoke(JRMPInvoker.java:359)
        at sun.reflect.GeneratedMethodAccessor83.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:261)
        at sun.rmi.transport.Transport$1.run(Transport.java:148)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
        at
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
        at java.lang.Thread.run(Thread.java:534)
```

La excepción de la consola de ClienteJugador nos da señal de que se ha producido un NullPointerException en la línea 864 de actualizaAtributosSimplesDeHabitacion. Eso se debe a que el método recibe como argumento un HabitacionBean nulo y esto siguiendo la traza se llega a la línea 222 que se hace lo siguiente:

getDao().store((agm.servidor.habitaciones.HabitacionBean) this);

Y este argumento es el que debe ser null para que se produzca ese NullPointerException por lo que de alguna forma la instancia desde donde se ejecuta store es null pero, ¿cómo puede ser esto posible? ¿Java permite algo así?
Se añadió al método store de HAbitacionDAOImpl el siguiente trozo de código para depurar justo el momento en el cuál llega a este método un argumento de tipo null y así observar la traza y el valor de las variables.
Arrancamos entonces el servidor en modo “Debug” con ese punto de interrupción y empezamos a mover un jugador de habitación a habitación hasta que se interrumpa la ejecución.

```
/**
     * @author Juan Carlos Castromil
     * @version 1.0 07/2007
     * Se almacena el ejb en BD
     * @param ejb
     */
    public void store(HabitacionBean ejb) throws EJBException {
       if (ejb == null) {
              log.info(“Se analizan las posibles pistas del error");
       }
       …
```

Conseguimos dar de nuevo con el error, pero lo curioso es que no se paró en el método store después de añadir el trozo de código con el punto de interrupción antes mostrado.
Entonces la única explicación posible es que realmente el argumento HabitacionBean del método actualizaAtributosSimplesDeHabitacion se "desconfigure" de alguna forma y se pase como null desde store.
Pero el propio método "actualizaAtributosSimplesDeHabitacion" hace varias llamadas a métodos de su argumento HabitacionBean antes de la línea en la que salta excepción y en esas no saltan. Para poder obtener más datos movemos el trozo de código que añadimos para depurar de "store" a "actualizaAtributosSimplesDeHabitacion" y así suponemos que cuando justo se fuera a producir la excepción se detendría la ejecución. El código queda en ese método de esta forma:

```
/**
     * @author Juan Carlos Castromil
     * @version 1.0 07/2007
     * Dada cierta habitacion actualizamos sus datos en BD
     * @param conn es la conexion con la cual se van a realizar las operaciones
     * sobre BD
     * @param ejb representa a la habitación sobre la que se van a actualizar
     * los datos
     */
    public void actualizaAtributosSimplesDeHabitacion(Connection conn,
            HabitacionBean ejb) {
       if (ejb == null) {
               log.info("Se analizan las posibles pistas del error");
       }
       …
```

Y volvemos a depurar de nuevo…

Vuelve a ocurrir la misma excepción después de un rato jugando y la misma traza pero la ejecución no se ha parado. La línea en la que se muestra violación es la siguiente:

ps.setString(count++, ejb.getIdHab().trim());

Entonces para que se lance un NullPointerException en esta línea y no sea por ejb debería ser o bien por ps o bien por getIdHab.
Si fuera ps, en ese mismo método debería haber saltado la violación en algunas líneas anteriores así que descartamos esa posibilidad.
Entonces debe ser por el id del ejb así que cambiamos el código que teníamos para observar la depuración para que se compruebe esto en vez de que ejb fuera null.
Se cambia entonces por

```
/**
     * @author Juan Carlos Castromil
     * @version 1.0 07/2007
     * Dada cierta habitacion actualizamos sus datos en BD
     * @param conn es la conexion con la cual se van a realizar las operaciones
     * sobre BD
     * @param ejb representa a la habitación sobre la que se van a actualizar
     * los datos
     */
    public void actualizaAtributosSimplesDeHabitacion(Connection conn,
            HabitacionBean ejb) {
       if (ejb.getIdHab() == null) {
               log.info("Se analizan las posibles pistas del error");
       }
       …
```

Y se vuelve a ejecutar AGM para ver si esta vez la ejecución llega a detenerse en el punto de interrupción añadido en la comprobación que hemos añadido…

Se consigue que se detenga la ejecución en este punto pero no se dan muchas pistas ya que la traza que se observa es tan solo la que se ejecuta desde el contenedor para llamar al método “store” de HabitacionBMP. El ejb sobre el que se ejecuta el método store tiene atributos a null y entre ellos el idHab por lo que se produce la excepción más tarde en la ejecución.
Un detalle curioso es que si se para la ejecución justo en el if que se ha añadido anteriormente, el mensaje inesperado “Primero se debe ejecutar la aplicación ClienteAdministrador” ya ha aparecido. Este mensaje se muestra como consecuencia de un NamingException en el método subscribeA de ClienteJugador. Este NamingException se produce como consecuencia de la búsqueda del topicString pasado como argumento al método así que de alguna forma el topicString no existe en ese momento. ¿Cuál puede ser la causa de que ahora no exista cuando en un momento anterior al error ya se estuvo en esa habitación? Quizá entonces poniendo un punto de interrupción en el bloque catch que captura la NamingException se podrá sacar más información sobre qué ha ocurrido.

Otro detalle a la hora de detenerse la ejecución es que el jugador en BD aparece aún en la antigua habitación. Es decir, aún no se ha completado la transacción que se encarga de cambiar el idHab del personaje jugador en la tabla personajesJ.

- **Explicación del origen del problema**:..

- **Solución aplicada**:

3. Mejora de la jugabilidad en AGM

3.1. Introducción

Una vez que se consiguió rearrancar la aplicación se puso de manifiesto que la interfaz gráfica dejaba bastante que desear y que en poco o nada se parecía a la de la primera versión del juego que, ejecutándose bajo MS-DOS, lucía mucho más intuitiva y amigable. De este modo se estableció como uno de los objetivos mejorar la jugabilidad de la AGM llevando a cabo una labor de reingeniería hasta conseguir replicar aquella primera interfaz.

Tras comprobar que la documentación sobre todo lo relacionado con la interfaz gráfica era casi inexistente y que ni las clases ni los métodos involucrados aparecían comentados se estableció un segundo objetivo: crear dicha documentación y comentar el código adecuadamente de tal forma que la interfaz gráfica, y el modo en que se relaciona con el resto de la aplicación, fuera más fácil de comprender y modificar en un futuro (cabe señalar que esta idea de facilitar la labor a nuestros sucesores ha sido predominante en nuestro proyecto dados los problemas con que nos encontramos desde un principio).

El tercer objetivo sería continuar con la integración de la versión animada que se empezara en el curso 2004/05, incluyéndose un estudio de su situación actual tras los cambios acometidos en la interfaz. Para finalizar, las mejoras que no han podido llevarse a cabo por falta de tiempo pero que podrían acometerse en el futuro han sido incluidas en el último apartado.

3.2. Cambios en la interfaz gráfica

3.2.1. Introducción

Como ya se ha comentado uno de los objetivos sería replicar la interfaz gráfica de la versión MS-DOS. Para ello, partiendo de la situación descrita en el punto 3.2.2., se realizó la labor de reingeniería comentada en el punto 3.2.3., hasta llegar a la interfaz descrita en el punto 3.2.4.

En esos tres puntos se contempla cómo se abordó el problema de una manera global, reservándose los puntos 3.2.5. y 3.2.6. para comentar los cambios concretos que se llevaron a cabo tanto a nivel de código (en cada uno de las clases involucradas) como a nivel visual (en cada uno de los componentes).

3.2.2. Así era la interfaz original

Los siguientes *snapshots* muestran el aspecto que presentaba la interfaz gráfica tal como la encontramos. En ellas se han señalado los principales componentes del juego:

Inventario: muestra los objetos que el personaje va consiguiendo a lo largo del juego y que le servirán para ir completando las distintas misiones. Se comienza con la llave de la taquilla.

Chat: está formado por un panel de salida en el que el jugador escribe sus mensajes y un panel de entrada en el que se muestran tanto sus mensajes como los del resto de jugadores que están conectados.

Pantalla de Juego: en ella se muestran las distintas habitaciones que va recorriendo el jugador así como el resto de jugadores, personajes u objetos que hay en cada una de ellas.

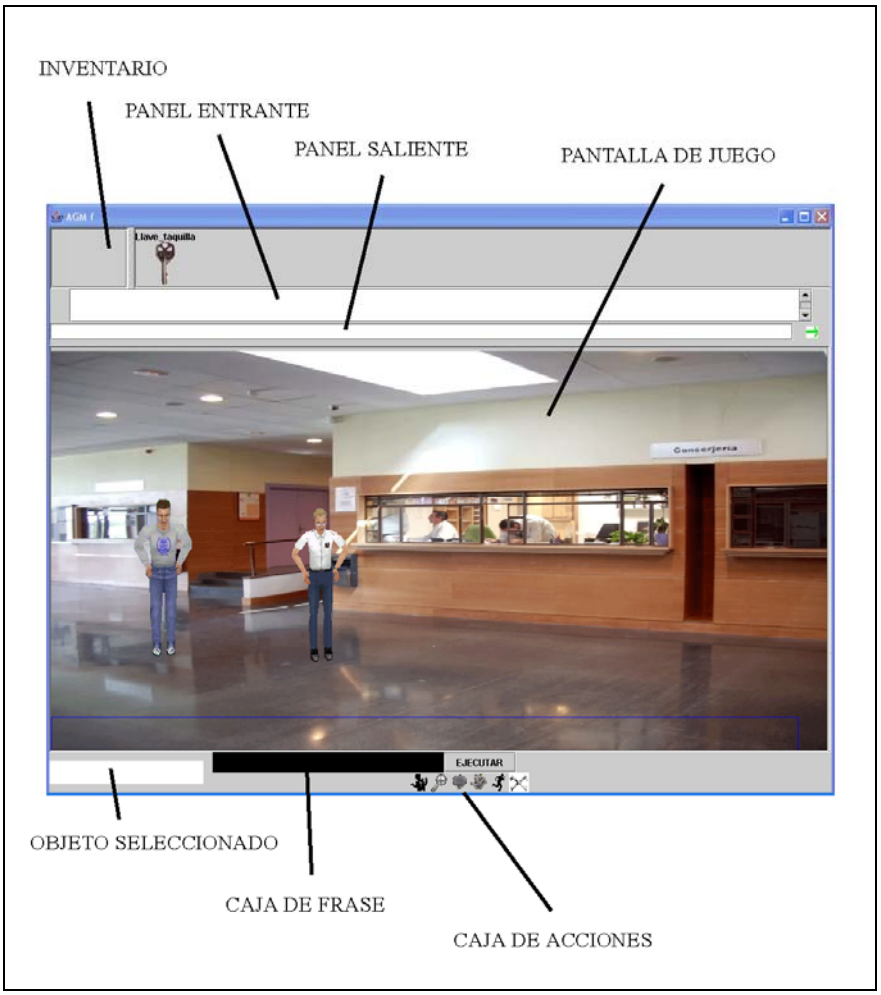
Consola no editable: se utiliza para mostrar tanto el resultado de algunas acciones (por ejemplo: mirar un objeto) como las conversaciones con los personajes del juego que lleva a cabo el jugador.

Objeto seleccionado: sirve para informar al jugador del objeto o personaje sobre el que pasa el puntero del ratón.

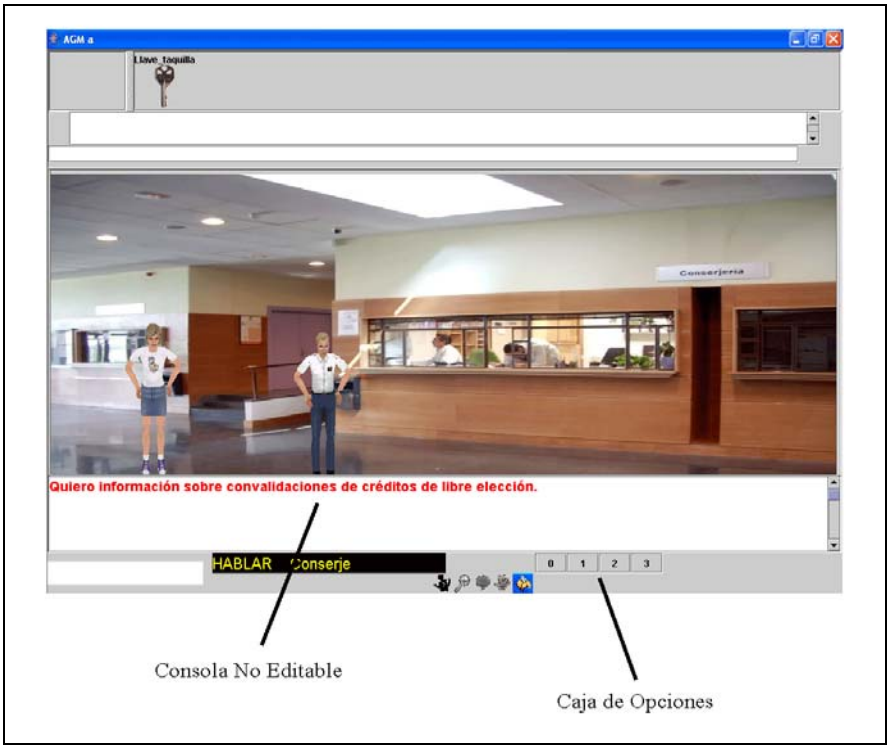
Caja de frase: en ella se muestra la acción que pretende realizar el jugador.

Caja de opciones: sirve para elegir la respuesta que el jugador le da al personaje cuando está teniendo lugar una conversación.

Caja de acciones: muestra las diferentes acciones que puede llevar a cabo el jugador (usar, mirar, hablar, coger, ir o salir del juego).



3. 1 Snapshot de la interfaz original



3. 2 Otra snapshot de la interfaz original

3.2.3. El proceso de reingeniería

Se trataba pues de valorar la interfaz gráfica de la versión MS-DOS y ésta, y plantear una nueva propuesta con la intención de dotar de la "amabilidad" de la primera las nuevas incorporaciones de la segunda (como el Chat para intercambiar mensajes con otros jugadores).

De este modo, para la nueva propuesta se pensó dividir la interfaz gráfica en tres partes claramente diferenciadas: la pantalla de juego en la mitad superior, un panel que agrupara la botonera de acciones, el inventario y el chat, en la parte inferior, y en medio un display en el que se mostrara la acción pretendida por el jugador.

Esta nueva propuesta también incluiría cambios en el panel de bienvenida al juego y en los de creación y conexión del personaje, pero por su independencia de la pantalla de juego en sí, serán tratados en el siguiente apartado.

Una vez elaborada la nueva propuesta se comenzaron a abordar los posibles cambios… Pero antes de introducir ninguno, y para evitar problemas no deseados que afectaran a la lógica del juego existente en ese momento, se hacía necesaria una comprensión total y absoluta del estado de la interfaz: saber cómo estaba estructurada, que hacía cada clase, cada método, quien les invocaba, qué función tenía cada componente, etc. Y no sólo de la interfaz sino también del modo en que ésta se relacionaba con el resto de la aplicación (ver apartado "Casos de uso").

Esta tarea no fue fácil en absoluto dado que no figuraba prácticamente nada ni en la documentación de los dos años anteriores ni en el código, y (respecto a la interfaz en sí) los conocimientos de AWT/Swing distaban mucho de ser los apropiados para abordar una de esta complejidad.

A esta inexistencia de información se unía lo caótico de la estructura tanto a nivel de código como a nivel visual. Por un lado clases, métodos y variables que no hacían nada, elementos redundantes, nombres no significativos, la no utilización de constantes, artificios engañosos (como el de hacer creer que era una aplicación válida para distintas configuraciones), uso inapropiado de componentes… Por otro lado, el caos en el código se plasmaba a nivel visual en una interfaz desordenada e irregular como se puede apreciar en los anteriores *snapshots*.

El siguiente paso fue por tanto tratar de entender lo que hacía cada clase, cada método y cada variable, añadiendo comentarios de línea, haciendo trazas, dibujando esquemas y bocetos, apuntando posibles mejoras, etc.

Así, se pudo llegar a una idea más o menos clara de cómo estaban organizados los componentes:

A nivel de componentes la Interfaz Grafica era un **JFrame** al que se le añadía un **JSplitPane** (separaConsolaEditableYPantalla).

Dicho **JSplitPan**e estaba formado a su vez por otro **JSplitPane** (separaInventYConsolaEditable) y un **Box** (cajaPantallaJuegoYConsolaNoEditableYFraseYAcciones).

El primero (separaInventYConsolaEditable) estaba formado por dos elementos de tipo **Box** a los que se añadían los componentes devueltos por los métodos *crearInventario* y *creaCajaChat*.

crearInventario devolvía un **JSplitPane** en cuyo componente derecho se añadía el componente devuelto por *getInventario*. El tipo de este componente era un **JScrollPane** en el que había un **Box** que a su vez era una sucesión de componentes Box formados por un **JLabel** y un **Jbutton**.

creaCajaChat devolvía un **Box** formado por un **JScrollPane** y otro **Box** compuesto de un **JTextField** y un **JButton**.

El segundo (cajaPantallaJuegoYConsolaNoEditableYFraseYAcciones) estaba formado por dos elementos de tipo **Box**: los devueltos por los métodos *creaCajaPantallaJuegoYConsolaNoEditable* y *crearCajaFraseYAccionesYOpciones*.

El **Box** que devolvía el primer método (creaCajaPantallaJuegoYConsolaNoEditable) estaba compuesto de un **JSplitPane** (separaPantallaJuegoYConsolaNoEditable) formado por dos elementos de tipo **Box**: los devueltos por los métodos *crearPantallaJuego* y *crearConsolaNoEditable*.

El **Box** devuelto por el primer método (*crearPantallaJuego*) era a su vez otro **Box** en el que se había añadido un **JScrollPane** al que a su vez se le había añadido un componente de tipo **Pantalla** (el cual extiende **JPanel**).

El **Box** devuelto por el segundo método (*crearConsolaNoEditable*) contenía un **JScrollPane** al que se había añadido un componente de tipo **JTextArea**.

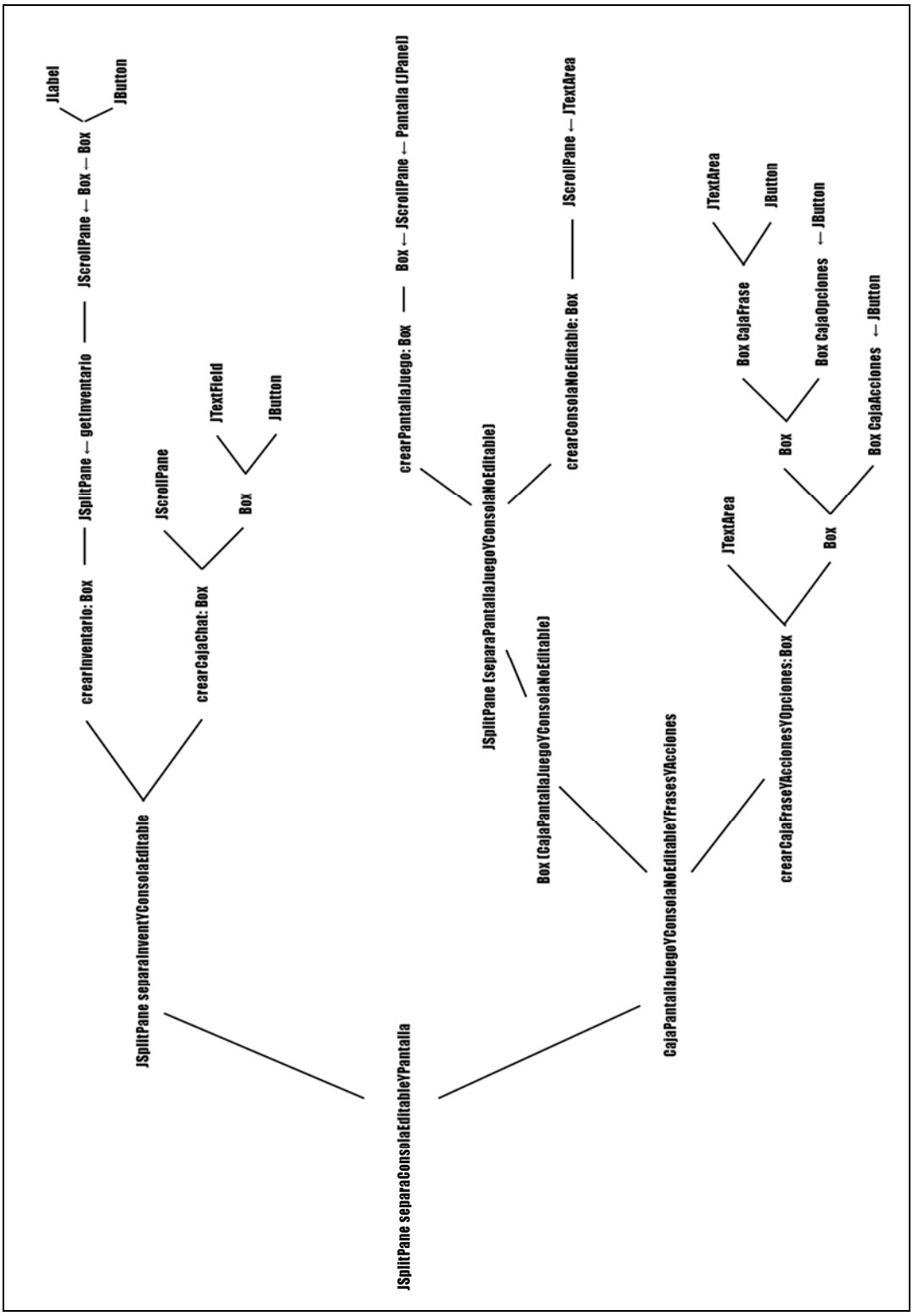
En cuanto al **Box** devuelto por *crearCajaFraseYAccionesYOpciones* estaba formado por un componente de tipo **JTextArea** y otro de tipo **Box**. Este último estaba formado por dos elementos de tipo **Box**: uno formado a su vez por dos elementos de tipo **Box**, devueltos por los métodos *crearCajaFrase* y *crearCajaOpciones*, y otro devuelto por el método *crearCajaAcciones*.

El **Box** devuelto por *crearCajaFrase* estaba formado por cuatro elementos de tipo **JTextArea** y un elemento de tipo **JButton**.

El devuelto por *crearCajaOpciones* estaba formado por un array de elementos de tipo **JButton**.

Por último el **Box** devuelto por *crearCajaAcciones* estaba formado por seis elementos de tipo **JButton**.

En el siguiente árbol se trata de plasmar todo este entramado de componentes:



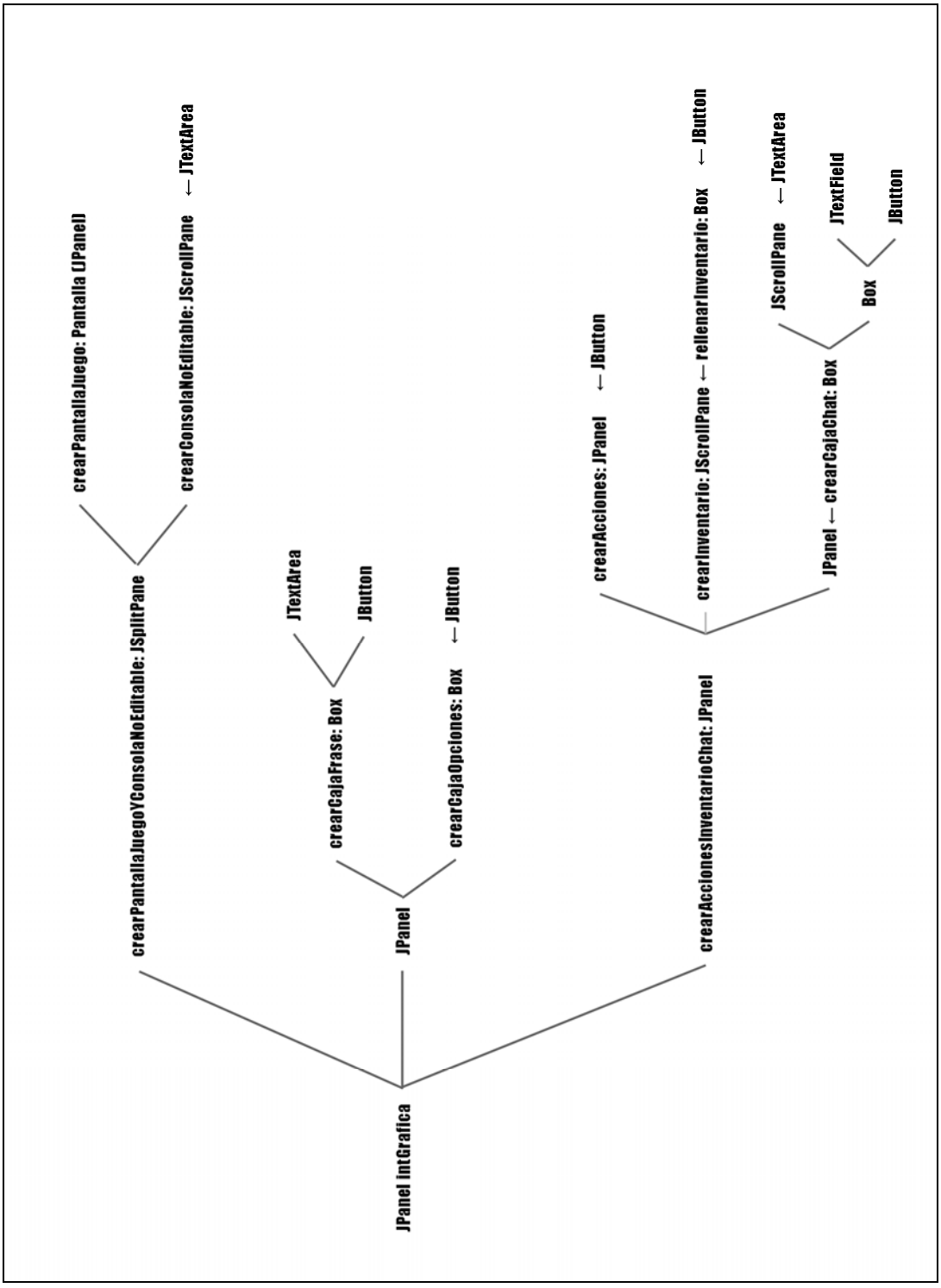
3.3 Árbol de componentes de la interfaz original

Era evidente el uso innecesario de algunos componentes (especialmente el apilamiento de contenedores **Box**) y el mal uso de otros (por ejemplo utilizar un **JButton** para mostrar una imagen pudiendo utilizar **JLabel**).

Partiendo de esto, y con la idea clara de la interfaz que queríamos conseguir se comenzó una labor de reingeniería de manera gradual realizando capturas y guardando

versiones "seguras" cada vez que se solucionaban los problemas que se iban produciendo a la hora de jugar.

Tras este lento y laborioso proceso, que será comentado para cada componente más adelante, se consiguió que el árbol de componentes de la interfaz actual quedara de la siguiente forma:



3.1. Árbol de componentes de la interfaz actual

Como puede apreciarse, a nivel de componentes la Interfaz Grafica es ahora un **JFrame** al que se le ha añadido un **JPanel** (intGrafica).

Dicho **JPanel** está formado por tres componentes: un **JSplitPane** (el devuelto por el método *crearPantallaJuegoYConsola*) y dos **JPanel**.

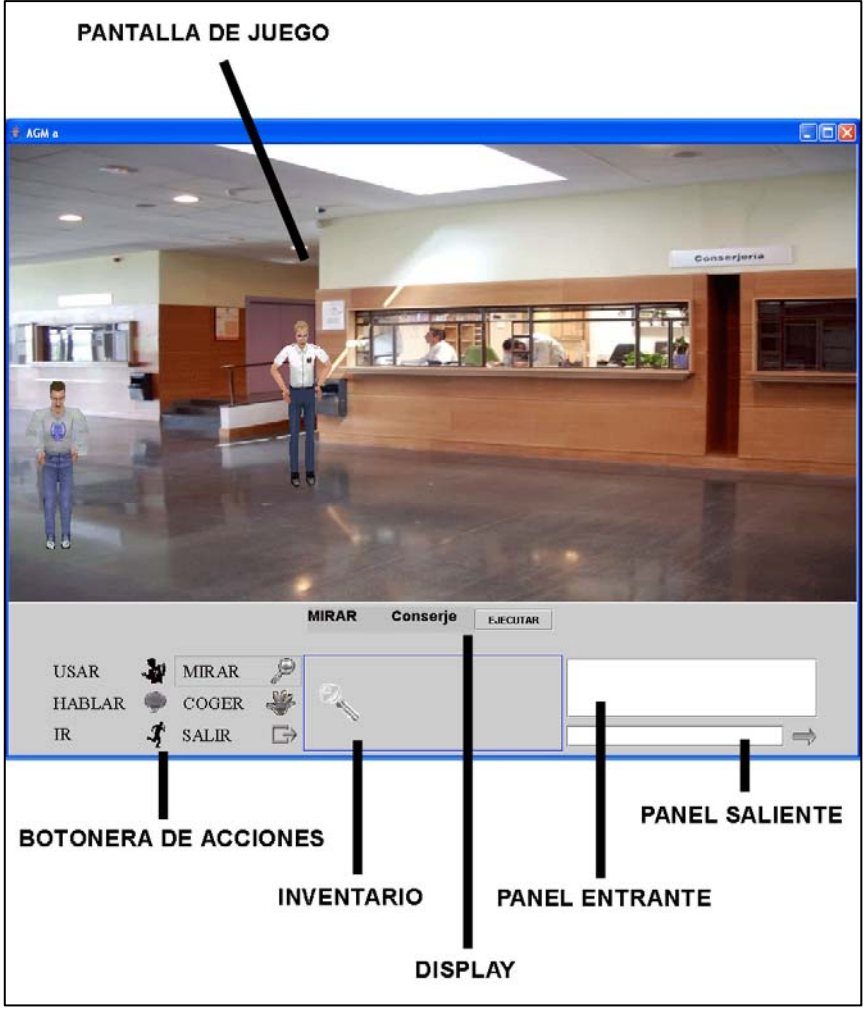
El **JSplitPane** está formado por dos componentes: el de tipo **Pantalla** que devuelve *crearPantallaJuego* y el de tipo **JScrollPane** que devuelve *crearConsola* (cabe señalar que en la nueva interfaz no tiene sentido hablar de ConsolaNoEditable pues sólo habrá una única consola).

El primer **JPanel** está formado por dos componentes de tipo **Box** (los devueltos por los métodos *crearCajaFrase* y *crearCajaOpciones*, que permanecen con la misma estructura que antes).

El segundo **JPanel** lo devuelve el método *crearAccionesInventarioChat* y agrupa tres componentes: el **JPanel** devuelto por *crearAcciones* (que está formado por seis elementos de tipo **JButton**), el **JScrollPane** devuelto por *crearInventario* (al que a su vez se le añade el **Box** formado por elementos de tipo **JButton** que devuelve *rellenarInventario*) y el **JPanel** que incluye el **Box** devuelto por *crearCajaChat* (con la misma estructura que antes).

3.2.4. Así es la nueva interfaz

El resultado de esta nueva composición es una interfaz gráfica más limpia y ordenada como puede apreciarse en las siguientes *snapshots*:



3.2. Snapshot de la interfaz actual

3.3. Otra snapshot de la interfaz actual

Como muestran el árbol y las *snapshots*, la interfaz de juego se ha dividido en tres partes claramente diferenciadas:

4. Los dos tercios superiores se han destinado a la pantalla de juego propiamente dicha donde se muestran los distintos objetos y personajes que existen en cada habitación.

5. El tercio inferior queda reservado a la botonera de las acciones, el inventario y el chat.

6. Separando ambas partes aparece un display en el que se muestra el objeto seleccionado o la acción que pretende realizar el jugador.

Esta nueva distribución de los componentes obedece a la interfaz de la versión MS-DOS a la que se hacía referencia al principio. Distribución que se buscaba replicar y a la que se ha incorporado el chat para la versión multijugador que existía en las versiones bajo Linux de años anteriores.

La pantalla de juego y la consola se han ajustado en anchura al tamaño máximo del panel que las contiene. El escenario del juego se ha agrandado en altura hasta ocupar exactamente las dos terceras partes superiores de la pantalla tal y como aparecía en la versión MS-DOS.

En cuanto al display se han agrupado todos los componentes (objeto seleccionado, caja de frase y botón Ejecutar) en un único panel y se ha situado, como en la versión MS-DOS, entre la pantalla de juego y los componentes del panel inferior.

Siguiendo con el ejemplo de la versión MS-DOS, la botonera de acciones se ha ampliado y situado en la esquina inferior izquierda de la pantalla. Las imágenes se han agrandado y, para algunas acciones, se han buscado unas más significativas y acordes con el resto de la interfaz. Así mismo, en lugar de un *tool tip* se ha acompañado a las imágenes del nombre de la acción para facilitar la jugabilidad.

Del mismo modo, el inventario, antes situado arriba en el interior de un incomprensible JSplitPane, aparece ahora junto a la botonera de acciones en la parte inferior de la pantalla. En la búsqueda de una mayor limpieza de la interfaz, la etiqueta que acompañaba al objeto, y que parecía redundante, ha sido sustituida por un *tool tip* que muestra el nombre del objeto al situarse encima el puntero del ratón.

En cuanto al chat, también se ha cambiado su ubicación situándolo en la esquina inferior derecha, para reservar así la parte central y superior a la pantalla de juego. Además se ha cambiado la imagen del botón por una mayor y más acorde con el resto de la interfaz, se han centrado los paneles entrante y saliente y se han separado un poco más. A las mejoras estéticas hay que añadirle una funcional que no es otra que la incorporación de un *scrollbar* horizontal y uno vertical que sólo serán visible cuando se necesite por la longitud del mensaje o por el tamaño de la conversación.

En cuanto a los paneles de bienvenida, creación del personaje y conexión, se han incorporado imágenes, botones, fondos y se ha jugado con colores, tamaños, fuentes y otros detalles hasta conseguir un aspecto más atractivo para el jugador.

A la sustitución de algunas imágenes por otras más idóneas en el conjunto de la interfaz, hay que añadir la unificación de criterios en las restantes y la introducción de optimizaciones en cuestiones tales como el formato y la resolución.

A las mejoras a nivel visual hay que sumar las mejoras en cuanto al código. Las líneas de código han disminuido drásticamente al utilizarse sólo las clases, métodos, variables y componentes estrictamente necesarios.

Como demostraremos en el siguiente apartado, se han eliminado hasta cinco clases, decenas de métodos, variables y componentes y cerca de mil líneas de código que o bien no servían para nada o bien se podían optimizar. Todo ello da cuenta de la situación encontrada al abordar la interfaz original, agravada como ya se ha comentado por la inexistencia de comentarios y documentación

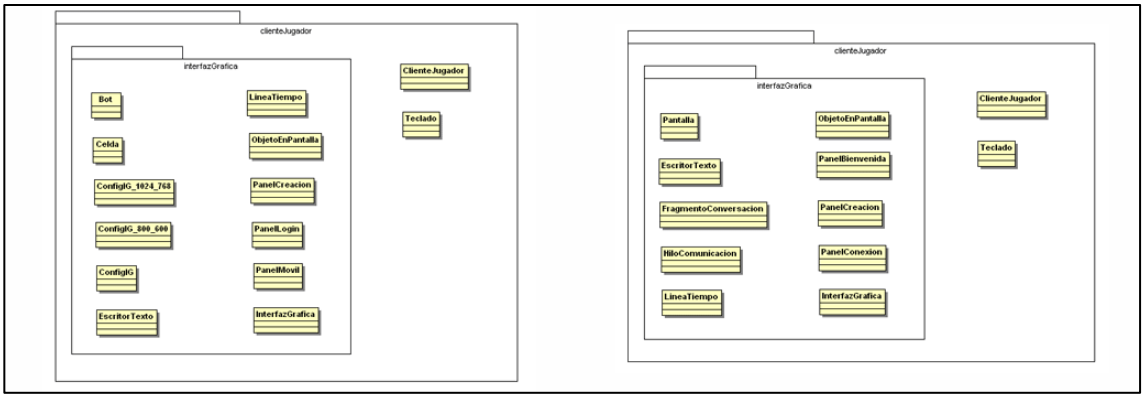
El resultado es una interfaz mucho más limpia y eficiente tanto visualmente como a nivel de código, que será más fácil de comprender y modificar en un futuro (por ejemplo para terminar de integrar la animación) pues se comenta el código adecuadamente (JavaDoc) y se explica todo detalladamente en esta documentación.

A continuación se tratará de explicar el proceso de reingeniería llevado a cabo hasta conseguir la nueva interfaz. Este análisis será realizado a dos niveles: por un lado a nivel de código (analizando los cambios en las clases y métodos involucrados) y por otro a nivel visual (analizando los cambios en cada uno de los componentes).

3.2.5. Análisis de clases

3.2.5.1. Paquete clienteJugador: Comparativa

A continuación se muestran las clases que contenía el paquete ClienteJugador con la interfaz original y las que contiene con la nueva interfaz.



3.4. Comparativa del paquete clienteJugador

Los cambios realizados han sido los siguientes:

1. Se han suprimido las tres clases relacionadas con la configuración de la pantalla (**ConfigIG**, **ConfigIG_1024_768** y **ConfigIG_800_600**).

2. Se ha suprimido la clase **PanelMovil**. Esta clase extendía **JPanel** y construía un objeto **PanelMovil** con dos **JLabel** ("ID" y "Mensaje") y sus correspondientes **JTextField**. Tras comprobar que no era utilizada por la aplicación se suprimió del paquete **ClienteJugador**.

3. Se ha suprimido la clase **Celda**. Esta clase extendía **JButton** y construía un objeto **Celda** con dos atributos de tipo int que recibía como parámetro. Todo apunta a que estaba concebida para capturar las coordenadas de los objetos en pantalla (antes de descubrir los métodos getX y getY del evento MouseEvent). Tras comprobar que no era utilizada por la aplicación se suprimió del paquete **ClienteJugador**.

4. Se ha añadido la clase **PanelBienvenida** (3.2.5.2.).

5. La clase **PanelCreacion** ha sufrido modificaciones (3.2.5.3.).

6. La clase **PanelLogin** pasa a llamarse **PanelConexion** (3.2.5.4.).

7. La clase **InterfazGrafica** ha sufrido modificaciones (3.2.5.5.).

8. La clase **Bot** pasa a llamarse **Pantalla** (3.2.5.6.).

9. La clase **ObjetoEnPantalla** ha sufrido modificaciones (3.2.5.7.).

10. La clase **EscritorTexto** ha sufrido modificaciones (3.2.5.8.).

11. Se ha añadido la clase **HiloComunicacion** (3.2.5.9.).

12. Se ha añadido la clase **FragmentoConversacion** (3.2.5.10.).

13. La clase **LineaTiempo** ha sufrido modificaciones (3.2.5.11.).

14. La clase **ClienteJugador** ha sufrido modificaciones (3.2.5.12.).

15. La clase **Teclado** sigue igual (3.2.5.13.).

A continuación se documenta cada una de las clases que contiene el paquete clienteJugador actual, explicando los cambios realizados en el caso de las ya existentes y el porqué de las nuevas que se han incorporado.

El orden que se sigue es el anterior. Se empieza por las tres clases de paneles (**PanelBienvenida**, **PanelCreacion** y **PanelLogin**), luego se analiza la clase que representa la interfaz (**InterfazGrafica**), luego las dos clases relacionadas con la pantalla de juego (**Pantalla** y **ObjetoEnPantalla**), despues las clases relacionadas con los mensajes que aparecen en la consola (**EscritorTexto**, **HiloComunicacion**,

FragmentoConversacion y **LineaTiempo**) y por último el oyente de la interfaz (**ClienteJugador**) y la clase **Teclado**.

3.2.5.2. La clase PanelBienvenida

Representa el panel de bienvenida que aparece cuando se ejecuta **ClienteJugador** y que permite al usuario la posibilidad de crear un personaje o de conectar uno ya existente para empezar a jugar.

Atributos

- **Color colorFondo = Color.YELLOW**
  Color de fondo del panel, establecido como constante para facilitar su posible cambio.

- **String rutaImagenes = "src/agm/clienteJugador/interfazGrafica/imagenes/"**
  Representa la ruta de la carpeta donde se encuentran todas las imágenes utilizadas en la aplicación.

Métodos

- **public PanelBienvenida()**
  Construye el panel de bienvenida (extiende **JPanel**) que será incrustado en un **JOptionPane** en el *main* de la clase **ClienteJugador**.
  Está formado por un título y una imagen, y ofrece al usuario la posibilidad de crear un personaje o de conectar uno ya creado y empezar a jugar.
  Su evolución visual se puede comprobar en el punto 3.2.6.1.

3.2.5.3. La clase PanelCreacion

Representa el panel de creación del personaje que aparece cuando se pulsa el botón "Crear Personaje" en el panel de bienvenida y que permite al usuario crear un personaje para poder jugar.

Atributos

- **JTextField id**
- **JTextField password**
- **JTextField nombre**
  Representan los campos donde se piden al usuario el id, el password y el nombre necesarios para crear el personaje.

- **JComboBox jugadores**
  Se utiliza para seleccionar uno de los cuatro posibles jugadores.

- **JLabel imagen**
  Se utiliza para mostrar la imagen de cada uno de los cuatro jugadores.

- **String rutaImagenes = "src/agm/clienteJugador/interfazGrafica/imagenes/"**
  Representa la ruta de la carpeta donde se encuentran todas las imágenes utilizadas en la aplicación.

- **Color colorFondo = java.awt.Color.YELLOW;**
  Color de fondo del panel, establecido como constante para facilitar su posible cambio.

Subclases

- **class OyenteCombo implements ItemListener**
  Oyente que va asociando al jugador seleccionado por el usuario su correspondiente imagen.

Métodos

- **public PanelCreacion()**
  Construye el panel de creación del personaje (extiende **JPanel**) el cual será incrustado en un **JOptionPane** dentro del método *solicitudCreacionPJ()* de la clase **ClienteJugador**.
  En este panel se le pide al usuario un ID, un password, un nombre y una imagen para crear el personaje con el que va a jugar.
  Su evolución visual se puede comprobar en el punto 3.2.6.1.

- **public String getNombre()**
- **public String getID()**
- **public String getPassword()**
- **public String getImagen()**
  Métodos invocados desde el método *solicitudCreacionPJ()* de **ClienteJugador** para obtener los valores introducidos por el usuario.

3.2.5.4. La clase PanelConexion

Representa el panel de conexión del personaje que aparece cuando se pulsa el botón “Conectar Personaje” en el panel de bienvenida y que permite al usuario conectar un personaje y empezar a jugar.

Atributos

- **JTextField id**
- **JTextField password**

Representan los campos donde se piden al usuario el id y el password necesarios para conectar el personaje.

- **JLabel imagen**

Se utiliza para mostrar la imagen de las llaves que aparece en el panel.

- **String rutaImagenes = "src/agm/clienteJugador/interfazGrafica/imagenes/"**

Representa la ruta de la carpeta donde se encuentran todas las imágenes utilizadas en la aplicación.

- **Color colorFondo = java.awt.Color.YELLOW;**

Color de fondo del panel, establecido como constante para facilitar su posible cambio.

Métodos

- **public PanelConexion()**

Construye el panel de conexión del personaje (extiende **JPanel**) el cual será incrustado en un **JOptionPane** dentro del método *solicitudConexionPJ()* de la clase **ClienteJugador**.
En este panel se le pide al usuario un ID y un password para conectar el personaje con el que va a jugar.
Su evolución visual se puede comprobar en el punto 1.2.6.1.

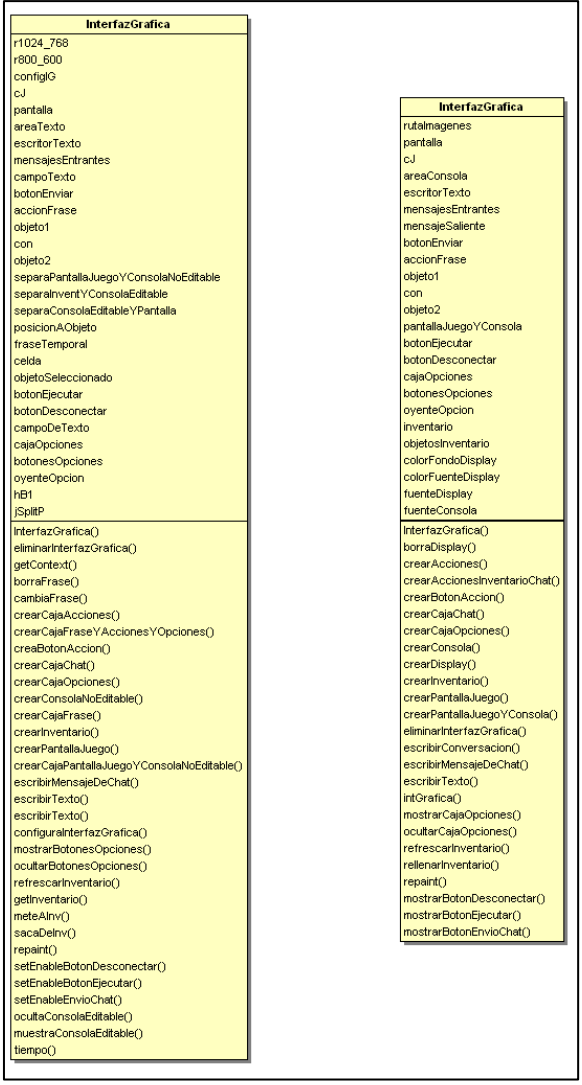
- **public String getID()**
- **public String getPassword()**

Métodos invocados desde el método *solicitudConexionPJ()* de **ClienteJugador** para obtener los valores introducidos por el usuario.

Su evolución visual se puede comprobar en el punto 3.2.6.1.

3.2.5.5. La clase InterfazGrafica

Implementa el GUI (interfaz gráfico de usuario) de la aplicación, formado por la pantalla de juego y la consola, el display, la botonera de acciones, el inventario y el chat.
En el proceso de reingeniería de esta clase han sido eliminadas cerca de 200 líneas de código correspondientes a variables y métodos que no se utilizaban, conservándose la misma funcionalidad.



3.5. Comparativa de métodos y atributos antes y después

Atributos

De las que había en la interfaz original se han eliminado las siguientes:

- **public final static int r1024_768 = 0;**
- **public final static int r800_600 = 1;**
- **private ConfigIG configIG;**
  Estas tres se han eliminado ya que, como se ha comentado anteriormente, el cambio en función de la resolución de la pantalla no estaba correctamente implementado (ver apartado “Trabajo futuro”).

- **private JTextArea objetoSeleccionado;**
  Aquí se mostraba el objeto o personaje sobre el que pasaba el ratón. En la nueva versión se ha suprimido este componente realizando su funcionalidad el JTextArea objeto1.

- **private JSplitPane separaInventYConsolaEditable;**
  Este componente representaba el JSplitPane que contenía el inventario y el chat en la interfaz original.

- **private JSplitPane separaConsolaEditableYPantalla;**

Este JSplitPane representaba el conjunto de la interfaz gráfica en su versión original (ahora de tipo JPanel).

- **private Hashtable posicionAObjeto;**
- **private boolean fraseTemporal;**

Estos dos atributos no se utilizaban. Ya no será necesario importar **java.util.Hashtable**.

- **private Celda celda;**
  Esta variable se utilizaba en la clase OyenteCelda, clase suprimida como se verá después.

Los atributos de la nueva interfaz (algunos renombrados con nombres más significativos) son las siguientes:

- **private static Pantalla pantalla;**
  Representa el escenario donde se desarrolla el juego (ver clase Pantalla).

- **private ClienteJugador cJ;**
  Representa el MessageListener sobre el que el usuario realizará todas las interacciones con el mundo virtual de AGM (ver clase ClienteJugador).

- **private static JTextArea areaConsola;**
  Representa la consola donde se escriben el resultado de algunas acciones y las conversaciones con los personajes no jugadores que lleva a cabo el jugador.

- **private static EscritorTexto escritorTexto;**
  Representa el componente anterior unido al JSplitPane que le contiene (ver clase EscritorTexto).

- **private JTextArea mensajesEntrantes;**
- **private JTextField mensajeSaliente;**
- **private JButton botonEnviar;**
  Estos tres atributos están relacionados con el chat para conversar con el resto de jugadores conectados. Representan el JTextField donde se escriben los mensajes, el botón que hay que pulsar para enviarlos y el JTextArea donde se muestra la conversación.

- **private JTextArea accionFrase;**
- **private JTextArea objeto1;**
- **private JTextArea con;**
- **private JTextArea objeto2;**
- **private JButton botonEjecutar;**
  Estos atributos están relacionados con el display. Representan los JTextField donde se muestra la acción que pretende realizar el jugador y el botón que tiene que pulsar para ejecutarla.

- **private static JSplitPane PantallaJuegoYConsola;**

Representa el JSplitPane donde se muestran el escenario del juego y la consola donde aparecen los mensajes.

- **private JButton botonDesconectar;**
  Representa el botón de la botonera de acciones con el que se sale de la aplicación.

- **private Box cajaOpciones;**
- **private JButton[] botonesOpciones;**
- **private OyenteOpcionMouse[] oyenteOpcion;**
  Estos tres atributos están relacionados con la botonera de opciones de respuesta que tiene para elegir el usuario cuando está hablando con un personaje no jugador.

- **private JScrollPane inventario;**
- **private Box objetosInventario;**
  Estos dos atributos representan el inventario: un Box que contiene los objetos incluido en un JScrollPane.

A las que añadimos:

- **private Color colorFondoDisplay= java.awt.Color.LIGHT_GRAY;**
  Color de fondo del display, elegido así para que toda la parte de abajo quede uniforme.

- **private Color colorFuenteDisplay= java.awt.Color.BLACK;**
  Color de la fuente del display, establecido también como constante para facilitar su posible cambio.

- **private Font fuenteDisplay= new Font("Arial", Font.BOLD, 18);**
  Representa el tipo de fuente para el display. En la versión original todas las fuentes se construían de la siguiente forma:

```
HashMap h = new HashMap();
h.put(TextAttribute.SIZE, new Float(20.0));
Font fuente = new Font(h);
```

Gracias a la nueva forma de construir las fuentes ya no será necesario importar las clases **java.util.HashMap** y **java.awt.font.TextAttribute**.

- **private Font fuenteConsola= new Font("Helvetica", Font.BOLD, 18);**
  Lo mismo que antes pero para la consola.

- **public static final String rutaImagenes = "src/agm/clienteJugador/interfazGrafica/imagenes/";**
  Representa la ruta de la carpeta donde se encuentran todas las imágenes utilizadas en la aplicación.

Subclases

De las que había en la interfaz original se han eliminado las siguientes:

- **private class OyenteCelda implements ActionListener**
  No se utilizaba, debía capturar las coordenadas de los objetos en pantalla antes de descubrir los métodos getX y getY del evento MouseEvent. Su código era el siguiente:

```
    private class OyenteCelda implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            boolean visible = false;
            ObjetoEnPantalla objeto = null;
            objeto = pantalla.getObjeto(((Celda) ev.getSource()).getFila(),
                    ((Celda) ev.getSource()).getColumna(), configIG
                            .getAltoBotonMatriz(), configIG
                            .getAnchoBotonMatriz());
            if (objeto != null)
                cambiaFrase(objeto);
        }
    }
```

- **private class OyenteMouseInventario implements MouseListener**
  El inventario tenía dos oyentes: **OyenteInventario** y **OyenteMouseInventario**. El código de **OyenteMouseInventario** era el siguiente:

```
    private class OyenteMouseInventario implements MouseListener {
        private ObjetoEnPantalla o;

        public OyenteMouseInventario(ObjetoEnPantalla o) {
            this.o = o;
        }
        public void mouseEntered(MouseEvent ev) {
            objetoSeleccionado.setText(o.getNombre());
        }
        public void mouseClicked(MouseEvent e) {
        }
        public void mouseExited(MouseEvent e) {
            objetoSeleccionado.setText("");
        }
        public void mousePressed(MouseEvent e) {
        }
        public void mouseReleased(MouseEvent e) {
        }
    }
```

  Como puede apreciarse, implementaba los métodos *mouseEntered* y *mouseExited* del panel del inventario (funcionalidad que fue sustituida por un *setToolTipText*). Los métodos *mousePressed* y *mouseReleased*, llamados al presionar y liberar el ratón (MouseMotion), no se utlizaban. El método *mouseClicked* podría sustituirse por el *actionPerformed* de **OyenteInventario**. Por estas razones se eliminó esta clase dejando un único oyente para el inventario. A consecuencia de esto ya no será necesario importar **java.awt.event.MouseListener**.

Las subclases de la nueva InterfazGrafica (algunas renombradas con nombres más significativos) son las siguientes:

- **private class OyenteOnClose extends WindowAdapter**

Implementa el método *windowClosing* llamado al pretender cerrar la ventana con el aspa de la esquina superior derecha. Si se está conversando con un personaje no jugador se mostrará un mensaje de fracaso por medio de un JOptionPane. En caso contrario se invocará al método *solicitudDesconexionPJ* de **ClienteJugador**.

- **private class OyenteInventario implements ActionListener**
  Cuando se hace click sobre un objeto del inventario se llama al método *actionPerformed* de esta clase, el cual se encargará de mostrar por pantalla el objeto del inventario en el campo objeto1 del display.

- **private class OyenteDesconectar implements ActionListener**
  Cuando se pulsa el botón Desconectar se llama al método *actionPerformed* de esta clase. Del mismo modo que en **OyenteOnClose**, si se está conversando con un personaje no jugador se mostrará un mensaje de fracaso por medio de un JOptionPane, invocándose en caso contrario al método *solicitudDesconexionPJ* de **ClienteJugador**.

- **private class OyenteAccion implements ActionListener**
  Es el oyente de la botonera de acciones. Cuando se pulsa una acción se ejecuta el método *actionPerformed* que se encarga de mostrarla en el display. En el caso de la acción "usar" se mostrará también el campo "con" y se hará visible el objeto2.
  A este oyente se le ha añadido el código necesario para implementar el cambio de puntero:

```
ImageIcon iconoPuntero = new ImageIcon(rutaImagenes+idAccion+"puntero.gif");
Image imagenPuntero = iconoPuntero.getImage();
Cursor cursor = Toolkit.getDefaultToolkit().createCustomCursor(imagenPuntero,new
Point(0,0),idAccion);
pantallaJuegoYConsola.setCursor(cursor);
```

  Para implementar este cambio hubo que importar las clases **Cursor**, **Toolkit** y **Point** (todas ellas presentes en el paquete **java.awt**). Las páginas consultadas fueron las siguientes:
  java.sun.com
  www.lawebdelprogramador.com
  www.koders.com

El cambio de puntero funciona de la siguiente forma: cuando se pulsa una acción se carga la imagen asociada (ejemplo: MIRARpuntero.gif) y se construye el objeto **Cursor** con dicha imagen. Por último se añade dicho cursor a su ámbito de visibilidad que no será otro que el JSplitPane formado por la consola y la pantalla de juego. Se podía haber hecho que el puntero retomase su aspecto original después de cada acción incluyendo la siguiente línea:

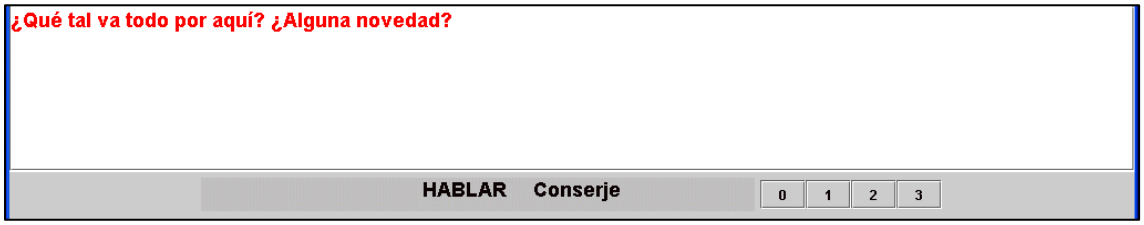
```
pantallaJuegoYConsola.setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
```

Pero se ha preferido que el puntero conserve la imagen de la última acción ejecutada y que no cambie hasta que no se pulse la acción siguiente. Para su efecto visual se aconseja ver el apartado 3.2.6.4. y el video Flash adjuntado a esta memoria.

- **public class OyenteEjecutar implements ActionListener**
  Cuando se pulsa el botón Ejecutar se llama al método *actionPerformed* de esta clase. En este método se capturan los atributos accionFrase, objeto1 y objeto2 del display, se actualizan los atributos idAccion, objeto1IdFrase y objeto2IdFrase de ClienteJugador y se llama a su método *ejecutaAccion*.

- **private class OyenteOpcion extends MouseAdapter**

Esta clase se encarga de mostrar en la consola las posibles respuestas que puede elegir un jugador cuando conversa con un personaje no jugador. Para ello se muestra un array de botones en el display (ver figura), cada uno de los cuales tiene asociado su correspondiente oyente para mostrar su correspondiente respuesta.



3.6. Aspecto de la consola durante una conversación

Gracias a los métodos *mouseEntered* y *mouseExited* aparece o desaparece la consola. El método *mouseClicked*, ejecutado al hacer click sobre una opción, ocultará la consola y evocará al método *teRespondo* de **ClienteJugador** pasándole como parámetro la opción elegida por el jugador:

```
public void mouseClicked(MouseEvent e) {
    ocultarBotonesOpciones();
    cJ.teRespondo(opcionElegida);
}
```

Esta clase no ha sido modificada por falta de tiempo aunque se apuntan algunas posibles mejoras en torno a estas conversaciones en el apartado “Trabajo futuro”. Para entender el mecanismo de las conversaciones se aconseja ver el caso de uso “Hablar con un personaje”.

- **private class OyentePantallaJuego extends MouseInputAdapter**
  Como su nombre indica es el oyente de la pantalla de juego.
  En la interfaz original, cuando se pasaba el ratón por encima de un personaje u objeto de la pantalla se mostraba su nombre en el campo objetoSeleccionado y cuando se hacía click sobre él se mostraba en el campo objeto1.
  En la nueva interfaz, y siguiendo el ejemplo de la versión MS-DOS, se ha suprimido el campo objetoSeleccionado de tal modo que cuando se pasa el ratón por encima de un personaje u objeto de pantalla se muestra directamente en el campo objeto1 del display (excepto si estaba seleccionada la acción “usar” que se mostrará en objeto2).
  Este cambio ha supuesto la transformación del método mouseMoved y la eliminación del método mouseClicked que había en la interfaz original y cuyo código se muestra a continuación:

```
        public void mouseClicked(MouseEvent e) {
            boolean visible = false;
            ObjetoEnPantalla objeto = null;
            objeto = pantalla.getObjeto(e.getX(), e.getY(), 1, 1);
            cambiaFrase(objeto);
        }
```

Métodos

Respecto a los métodos, de los que había en la interfaz gráfica original han sido eliminados han sido los siguientes:

- **public void meteAInv(ObjetoEnPantalla o)**
- **public void sacaDeInv(String idO)**

  Estos dos métodos se encargaban de meter o sacar del inventario el objeto que recibían como parámetro, pero no se les invocaba desde ningún sitio. Esa funcionalidad la llevaba a cabo **private JScrollPane getInventario(boolean generarInventFijo)** (ahora **private Box rellenarInventario()**).

- **private int tiempo(int length)**

  Este método tampoco se utilizaba. Su código era el siguiente:

```
    private int tiempo(int length) {
        return 5;
    }
```

- **private void muestraConsolaEditable()**
- **private void ocultaConsolaEditable()**

  En una primera versión del chat, la consola editable debía ser al inventario lo que la consola no editable a la pantalla de juego, esto es, el componente inferior de un JSplitPane que aparecía y desaparecía a lo largo del juego. Tras comprobar que ambos métodos no eran utilizados fueron eliminados (así como las referencias a una "consola editable", renombrando "consola no editable" como "consola"). El código era el siguiente:

```
    private void ocultaConsolaEditable() {
        separaInventYConsolaEditable.setDividerLocation(105);//105
    }

    private void muestraConsolaEditable() {
        separaInventYConsolaEditable.setDividerLocation(80);//80
        separaConsolaEditableYPantalla.setDividerLocation(155);//105
    }
```

- **private void cambiaFrase(ObjetoEnPantalla objeto)**

  La frase con la acción que pretende realizar el jugador está gestionada por una incomprensible máquina de estados que, en función del estado de la frase (0, 1 ó 2), muestra u oculta los diferentes JTextField y actualiza el atributo estadoFrase de ClienteJugador. Su código es el siguiente:

```
    private void cambiaFrase(ObjetoEnPantalla objeto) {

        if ((cJ.getEstadoFrase() == 0) || (cJ.getEstadoFrase() == 2)) {
            cJ.setObjeto1IdFrase(objeto.getIdObjeto());
            objeto1.setText(objeto.getNombre());
```

```
            if (cJ.getEstadoFrase() == 2) {
                con.setVisible(false);
                objeto2.setVisible(false);
                cJ.setObjeto2IdFrase(null);
            }
            cJ.setEstadoFrase(1);
        }

        else if ((cJ.getEstadoFrase() == 1)
                && accionFrase.getText().equals("USAR")) {
            if ((objeto != null)
                    & (cJ.getObjeto1IdFrase() != objeto.getIdObjeto())) {
                cJ.setObjeto2IdFrase(objeto.getIdObjeto());
                objeto2.setText(objeto.getNombre());
                con.setVisible(true);
                objeto2.setVisible(true);
                cJ.setEstadoFrase(2);
            }
        } else if (cJ.getEstadoFrase() == 1) {
            if (objeto != null) {
                cJ.setObjeto1IdFrase(objeto.getIdObjeto());
                objeto1.setText(objeto.getNombre());
                objeto1.setVisible(true);
            }
        }
    }
```

En la nueva interfaz queda pendiente la sustitución de está máquina de estados por una nueva implementación según la cual los oyentes de la pantalla de juego, de la botonera de acciones y del inventario se ocupen de mostrar o no los JTextField del display (para actualizar las propiedades de un componente no será necesario usar threads), invocándose (cuando se pulse el botón Ejecutar) a *ejecutarAccion* de **ClienteJugador** con los parámetros que haya en ese momento en el display: accion, objeto1 y objeto2.

- **public void escribirTexto(String texto, Color c, boolean esLaUltimaLinea)**

  Este método está relacionado con la forma en que se escribían las conversaciones con un personaje no jugador en la versión original. Su sustitución por el método escribirConversación se explica más adelante y responde al bug explicado en el apartado 2.5.2.

- **getContext()**

  Este método para obtener el contexto inicial fue suprimido ya que no era necesario en esta clase (sí lo será en ClienteJugador). Su código era el siguiente:

```
    private InitialContext getContext() throws NamingException {
        Hashtable props = new Hashtable();
        props.put(InitialContext.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
        props.put(InitialContext.PROVIDER_URL, "jnp://127.0.0.1:1099");
        InitialContext initialContext = new InitialContext(props);
        return initialContext;
    }
```

A consecuencia de esto no será necesario importar las clases **javax.naming.InitialContext** y **javax.naming.NamingException**.

De este modo, los métodos de la nueva interfaz, algunos renombrados con nombres más significativos, son los siguientes (ordenados top-down):

- **public InterfazGrafica(ClienteJugador cJ)**

  Constructor de la clase. Le pone un título al **JFrame**, inicializa cj con el **ClienteJugador** que recibe como parámetro, añade el **JPanel** devuelto por *intGrafica*, pinta la pantalla y arranca el escritor de texto.

- **public void eliminarInterfazGrafica()**

  Elimina el **JFrame** mediante un *dispose* cuando se confirma la desconexión del jugador.

- **public void repaint()**

  Llama al método *repaint* de **JFrame** y al de la pantalla (que extiende **JPanel**).

- **private JPanel intGrafica()**

  Mete la pantalla de juego y la consola en un panel, el display y la caja de opciones en otro, y la botonera de acciones, el inventario y el chat en otro. Por último los distribuye y los devuelve en un **JPanel**.

- **private JSplitPane crearPantallaJuegoYConsola(JPanel pantallaJuego, JScrollPane consola)**

  Devuelve un JSplitPane cuyo componente superior es la pantalla de juego (de tamaño 1024x550) y cuyo componente inferior será la consola (que se irá mostrando y ocultando a lo largo del juego).

- **public Pantalla crearPantallaJuego()**

  Devuelve la pantalla de juego (de tipo **Pantalla**) junto a su oyente.

- **private JScrollPane crearConsola()**

  Devuelve la consola donde se mostrarán los mensajes (una **JTextArea** dentro de un **JScrollPane**).

- **public void escribirTexto(String texto)**

  Utilizando la clase **EscritorTexto** muestra en la consola el texto que recibe como parámetro.

- **public void escribirConversacion(List comunicacion)**

  Utilizando las clases **EscritorTexto**, **HiloComunicacion** y **FragmentoConversacion** muestra en la consola el texto que recibe como parámetro.

  Los cambios en la pantalla se comentan en el apartado 2.5.2. Para entender el funcionamiento de la consola se aconseja ver también los casos de uso “Ejecutar una acción” y “Hablar con un personaje”.

- **private Box crearDisplay()**

  Devuelve el display (de tipo **Box**) donde se muestra la acción que pretende llevar a cabo el jugador acompañado del botón Ejecutar.

- **public void borraDisplay()**

Borra la frase que hubiera en el display.

- **public void mostrarBotonEjecutar(boolean visible)**

Muestra o no el botón Ejecutar en función del booleano que recibe como parámetro. Se ocultará mientras dure una conversación con un personaje no jugador para que ésta no se interrumpa.

- **public Box crearCajaOpciones()**

Devuelve la caja de opciones de respuesta (de tipo **Box**) que aparece cuando se conversa con un personaje no jugador.

- **public void mostrarCajaOpciones(LinkedList opciones)**

Muestra la caja de opciones de respuesta. Para ello itera sobre las opciones que recibe como parámetro (de tipo LinkedList) mostrando el correspondiente botón y asignándole el texto de respuesta a su oyente.

- **public void ocultarCajaOpciones()**

Oculta la caja de opciones iterando sobre cada uno de sus botones.

Los cambios en el display se comentan también en el apartado 3.2.6.3. Para entender el funcionamiento de la caja de opciones se aconseja ver el caso de uso "Hablar con un personaje".

- **private JPanel crearAccionesInventarioChat()**

Devuelve el **JPanel** formado por la botonera de acciones, el inventario y el chat.

- **public JPanel crearAcciones()**

Devuelve el **JPanel** formado por los seis botones de acciones (usar, mirar, hablar, coger, ir a, y salir).

- **private JButton crearBotonAccion(String accion, boolean asociarOyenteAccion)**

Devuelve el **JButton** de la acción que recibe como parámetro. Se asocia un oyente de tipo **OyenteAccion** a todos los botones excepto al botón Desconectar que tiene el suyo propio (de tipo **OyenteDesconectar**).

Cabe señalar que este método recibía antes dos parámetros: uno de tipo **Font** que ya no se utiliza y otro de tipo **String** que representaba el formato de la imagen de cada botón y que ha sido suprimido al uniformar todas las imágenes simples a GIF.

- **public void mostrarBotonDesconectar(boolean visible)**

Muestra o no el botón Desconectar en función del booleano que recibe como parámetro. Se ocultará mientras dure una conversación con un personaje no jugador para que ésta no se interrumpa.

Los cambios en la botonera de acciones se comentan también en el apartado 3.2.6.4.

- **private JScrollPane crearInventario()**

Devuelve el **Box** que contiene los objetos del inventario dentro de un **JScrollPane**.

- **public void refrescarInventario()**

Se le llama desde **ClienteJugador** cuando se mete un nuevo objeto o se saca uno que ya estuviera.

- **private Box rellenarInventario()**

Este método itera sobre los objetos del jugador (que recibe a través de **ClienteJugador**) y los va añadiendo al **Box** contenido por el inventario (**JScrollPane**).

Además de la eliminación de los métodos *MeteAInv* y *sacaDeInv* comentada con anterioridad, cabe señalar que el cambio en el fondo (no distinción entre un inventario fijo y otro no-fijo) y en la forma (eliminación del **JSplitPane**) en que estaba concebido el inventario original han permitido reducir a la mitad el número de líneas de código de este método (antes *getInventario*). Estos cambios se comentan en el apartado 3.2.6.5.

Para entender la forma en que se actúa sobre el inventario se aconseja ver el caso de uso "Ejecutar una acción".

En el apartado "Trabajo futuro" se apunta una posible mejora en torno al inventario consistente en implementar el cambio de cursor al hacer click sobre un determinado objeto (como ocurría en la versión MS-DOS).

- **private Box crearCajaChat()**

Devuelve el chat (de tipo **Box**) donde se llevan a cabo las conversaciones con el resto de jugadores conectados. Está formado por un panel entrante (**JTextArea** contenida en **JScrollPane**) y un panel saliente (un **Box** que contiene un **JTextField** y un **JButton** "Enviar").

- **public void escribirMensajeDeChat(String nombrePJ, String mensaje)**

Se le llama desde **ClienteJugador** al procesar un objeto de tipo MENSAJE_CHAT. Muestra en el panel entrante dicho mensaje precedido del nombre del jugador que lo escribió.

- **public void mostrarBotonEnviar(boolean visible)**

Muestra o no el botón Enviar en función del booleano que recibe como parámetro. Se ocultará mientras dure una conversación con un personaje no jugador para que ésta no se interrumpa.

Los cambios en el chat se comentan en el apartado 3.2.6.6. Para entender la forma en que funciona se aconseja ver el caso de uso "Hablar con otros jugadores".

3.2.5.6. La clase Pantalla

Extiende **JPanel** e implementa la pantalla de juego donde se van mostrando las distintas habitaciones con sus respectivos jugadores, personajes no jugadores y objetos.
Esta clase (antes llamada Bot) tiene su explicación en que para meter una imagen en un panel es necesario crear una clase que herede de **JPanel** y sobreescribir su método *paintComponent* (www.infosintesis.net).

Atributos

- **private ClienteJugador cJ**
  Representa el MessageListener sobre el que el usuario realizará todas las interacciones con el mundo virtual de AGM (ver clase **ClienteJugador**).

- **private int altoImagenFondo=550**
- **private int anchoImagenFondo=1024**
  Dimensión de la imagen de las habitaciones (fijada para una resolución 1024x768 px).

- **private Image imagenFondo**
- **private String nombreImagenFondo**
  Representan la imagen de fondo (en formato JPEG) y su nombre.
  Cabe señalar que JPEG es el formato óptimo para imágenes del mundo real (como es el caso de las habitaciones), funcionando mejor el formato GIF para dibujos simples (como es el caso de los jugadores, personajes, objetos, puntero y botones).
  Con el cambio de formato de los personajes de PNG a GIF se consiguió una reducción del peso cercana al 80% (fue necesario modificar el método *creaObjetosEnPantalla* de **ClienteJugador**).
  Para cuestiones de formato de imágenes se consultaron los siguientes enlaces:
  www.unav.es
  www.tejedoresdelweb.com

- **public static final String rutaImagenes = "src/agm/clienteJugador/interfazGrafica/imagenes/"**
  Representa la ruta de la carpeta donde se encuentran todas las imágenes utilizadas en la aplicación.

Métodos

- **public Pantalla (ClienteJugador cJ, int anchoImagenFondo, int altoImagenFondo)**

  Construye el objeto de tipo Pantalla con la imagen de fondo cuyo nombre recibe de **ClienteJugador** y la escala al ancho y alto necesario (en este caso fijado a una resolución 1024x768 px).

- **public void paint(Graphics g)**

  Mediante el método *drawImage* y la lista ObjetosEnPantalla del jugador, pinta la imagen de fondo y los distintos objetos, personajes y jugadores que se encuentran en dicha habitación.
  Las líneas de código de este método (que en su origen debió de albergar numerosas pruebas) han sido reducidas a la mitad. Como consecuencia de esta reducción se hizo innecesario importar las clases **Font**, **TextAttribute** y **HashMap**.
  A continuación se muestra a modo de ejemplo un trozo de código que se quitó:

```
Color c1 = g.getColor();
g.setColor(Color.RED);
HashMap h = new HashMap();
h.put(TextAttribute.SIZE, new Float(20.0));
Font fuente = new Font(h);
g.setFont(fuente);
g.drawString(cJ.getTextoConversacion(), 100, 300);
g.setColor(c1);
g.setColor(Color.RED);
//g.drawRect(0,400,780,120);//posicionamiento de pies en el hall
//g.drawRect(518,400,486,120);//posicionamiento de pies en el
// BAï¿½O
g.drawRect(0, 480, 970, 45);//posicionamiento de pies en el pasillo
//g.drawRect(0,238,970,285);//posicionamiento de pies en el pasillo
//g.drawRect(250, 370,585,150);//posicionamiento de pies en el
// pasillo300.300-900.500
```

- **public ObjetoEnPantalla getObjeto(int x, int y)**

  Devuelve el objeto que se encuentra en las coordenadas (x,y) de la pantalla de juego (siempre y cuando pertenezca a la lista ObjetosEnPantalla del jugador).
  En este método fueron suprimidos los enteros altoBoton y anchoBoton que recibía como parámetro ya que al ser invocado (desde **OyentePantallaJuego**) siempre se fijaban ambos valores a 1.

- **public boolean solapa(int x1, int y1, int x2, int y2)**

  Es invocado por el método anterior. Devuelve true si se solapan los dos objetos cuyas coordenadas recibe como parámetro.

3.2.5.7. La clase ObjetoEnPantalla

Representa la clase de los objetos que pueden mostrarse en pantalla. Estos pueden ser objetos no personajes, personajes no jugadores y personajes jugadores.
En la nueva versión, esta clase ya no implementa la clase **Cloneable** ya que no se utiliza el método *clone* en ningún sitio de la aplicación.
Al unificar el formato de todos los objetos en pantalla a **gif** se suprimió el atributo formato de esta clase y los métodos *setFormato* y *getFormato*. Este cambio sólo afecto a los métodos *rellenarInventario* (en **InterfazGrafica**), y *creaObjetosEnPantalla* y *procesaMetePJ* (en **ClienteJugador**). El motivo es que antes del cambio, las imágenes de los jugadores estaban en formato **png**.

Atributos

- **private String nombre**

Nombre del objeto.

- **private String estado**

Estado del objeto.

- **private String idObjeto**

Id del objeto.

- **private int tipoObjeto**

Tipo del objeto (0 si es un objeto no personaje, 1 si es un personaje no jugador y 2 si es un personaje jugador).

- **private int posX**
- **private int posy**

Posición del objeto.

- **private int alto**
- **private int ancho**

Alto y ancho del objeto.

- **private String nombreImagen**
- **private Image imagen**

Imagen del objeto y nombre de la imagen.

Métodos

- **public ObjetoEnPantalla(String nombre, String estado, String idObjeto, int tipoObjeto, int posX, int posY, int ancho, int alto, String nomIm)**

Constructor de la clase ObjetoEnPantalla.

- **public String getNombre()**
- **public void setNombre(String string)**

Métodos para acceder al nombre del objeto.

- **public int getAlto()**
- **public void setAlto(int i)**
- **public int getAncho()**
- **public void setAncho(int i)**

Métodos para acceder al ancho y alto del objeto.

- **public String getIdObjeto()**
- **public void setIdObjeto(String string)**

Métodos para acceder al id del objeto.

- **public int getPosX()**
- **public void setPosX(int i)**
- **public int getPosY()**

- **public void setPosY(int i)**

Métodos para acceder a la posición del objeto.

- **public String getEstado()**
- **public void setEstado(String string)**

Métodos para acceder al estado del objeto.

- **public int getTipoObjeto()**
- **public void setTipoObjeto(int i)**

Métodos para acceder al tipo del objeto.

- **public String getNombreImagen()**
- **public void setNombreImagen(String string)**
- **public Image getImagen()**
- **public void setImagen(Image imagen)**

Métodos para acceder a la imagen y al nombre de la imagen del objeto.

3.2.5.8. La clase EscritorTexto

Esta clase representa la consola donde se muestran los mensajes a lo largo del juego y fue modificada al arreglar el bug por el que no finalizaban, o se perdían mensajes, en algunas conversaciones.

Atributos

- **private String texto**

Representa la línea de texto que se va a escribir.

- **private JTextArea g**

Representa el área de texto donde se va a escribir.

- **private LinkedList lineas**

Representa la lista con todas las líneas que forman la conversación.

- **private JSplitPane p**

Representa el **JsplitPane** que contiene la pantalla de juego y la consola y será necesario para poder mostrar u ocultar ésta última.

- **private ClienteJugador cJ**

Representa el MessageListener sobre el que el usuario realizará todas las interacciones con el mundo virtual de AGM (ver clase **ClienteJugador**).

- **public static int contador = 0**

Lleva el orden en el cual deberían llamar los distintos hilos de la comunicación al método *escribeTexto*.

- **private volatile boolean ocupado = false**

Valdrá true mientras se esté ejecutando *escribeTexto*.

Respecto a los atributos, han sido suprimidos dos de tipo **JTextArea** que aparecían en la versión original y que no se utilizaban.

Métodos

- **public EscritorTexto(JTextArea g, JSplitPane p)**

Construye el objeto formado por el area de texto donde se va a escribir y el panel que la contiene, es decir la consola donde se muestran los mensajes a lo largo del juego.

- **public synchronized void escribeTexto(LineaTiempo lt)**

Añade a las lista de líneas la línea que recibe como parámetro.

- **private void muestraConsola ()**
- **private void ocultaConsola ()**

Métodos para mostrar u ocultan la consola.

- **public void run()**

Arranca el hilo encargado de escribir las conversaciones entre el jugador y el personaje no jugador.

- **private synchronized void quitaLinea()**

Se quita de la lista de líneas la primera.

- **private synchronized void escribeTexto(LineaTiempo lineaTiempo, boolean esLaUltimaLinea, java.awt.Color c) throws InterruptedException**

Escribe en la consola la línea que recibe como parámetro y se deja un tiempo en función de la longitud del mensaje a mostrar para que no se pueda escribir más texto hasta entonces y al usuario le de tiempo a leerlo.

- **public synchronized void escribeTexto(FragmentoConversacion fragmento, ClienteJugador cJ, int contadorL)**

Escribe en la consola el fragmento de la conversación (lo que dice seguido un personaje o jugador) que recibe como parámetro y, al igual que en el método anterior, se deja un tiempo en función de la longitud del mensaje a mostrar para que no se pueda escribir más texto hasta entonces y al usuario le de tiempo a leerlo.

3.2.5.9. La clase HiloComunicacion

Esta clase fue añadida al paquete **cliente.jugador.interfazGrafica** al arreglar el bug por el que no finalizaban, o se perdían mensajes, en algunas conversaciones.
Se encarga de arrancar un hilo aparte que va iterando sobre los distintos hilos de conversación de los que dispone.

Atributos

- **private static final Logger log = Logger.getLogger(HiloComunicacion.class)**

Representa el objeto con el cual se escriben las excepciones.

- **private EscritorTexto escritor**

Representa la consola donde se escribe el texto.

- **private FragmentoConversacion fragmento**

Representa lo que habla seguido cada personaje o jugador.

- **private ClienteJugador jugador**

Representa el MessageListener sobre el que el usuario realizará todas las interacciones con el mundo virtual de AGM (ver clase **ClienteJugador**).

- **private int contadorLocal**

Representa el orden en el cual deben llamar los distintos hilos de la comunicación al método *escribeTexto*.

Métodos

- **public HiloComunicacion (FragmentoConversacion fragment, EscritorTexto writer, ClienteJugador cliente, int orden)**

Construye un objeto de la clase **HiloComunicacion**.

- **public void run ()**

Llama al método *escribeTexto* de **EscritorTexto**.

3.2.5.10. La clase FragmentoConversacion

Esta clase fue añadida al paquete **cliente.jugador.interfazGrafica** al arreglar el bug por el que no finalizaban, o se perdían mensajes, en algunas conversaciones.
Representa el fragmento de la conversación que habla seguido cada personaje o jugador y sirve para encapsular todo lo relacionado con dichos fragmentos.

Atributos

- **private String frase**

Frase del fragmento que se va a escribir.

- **private Color color**

Color con el que se escribirá el fragmento

- **private boolean ultimaLinea**

Booleano para indicar si es la última línea del fragmento.

Métodos

- **public Color getColor()**
- **public void setColor(Color color)**

Métodos para acceder al color con el que se escribirá el fragmento.

- **public String getFrase()**
- **public void setFrase(String frase)**

Métodos para acceder a la frase del fragmento que se va a escribir.

- **public boolean isUltimaLinea()**
- **public void setUltimaLinea(boolean ultimaLinea)**

Métodos para acceder a esta propiedad del fragmento (si es la última línea o no).

3.2.5.11. La clase LineaTiempo

Esta clase está relacionada con la línea de texto que se va a mostrar en la consola.

Atributos

- **private String linea**

Esta variable representa la línea de texto que se va a mostrar en la consola.

- **private java.awt.Color color**

Representa el color de la línea.

- **private ClienteJugador cJ**

Representa el MessageListener sobre el que el usuario está realizando el intercambio de mensajes (ver clase **ClienteJugador**).

- **private boolean esLaUltimaLinea**

Indica si la línea que se va a mostrar es la última del mensaje o conversación.

- **private boolean esConversacion**

Indica si la línea que se está escribiendo forma parte de una conversación con un personaje o si, por el contrario, es un mensaje informativo del resultado de una acción.

Métodos

- **public LineaTiempo(String linea)**

  Constructor de la clase, inicializa sus atributos.

- **public String getLinea()**
- **public void setLinea(String string)**

Métodos para acceder a la línea de texto que se va a escribir en la consola.

- **public java.awt.Color getColor()**

Método para obtener el color de la línea de texto que se va a escribir.

- **public ClienteJugador getCJ()**

Método para obtener el oyente sobre el que el jugador está realizando el intercambio de mensajes.

- **public boolean isEsLaUltimaLinea()**

Devuelve true si la línea que se va a mostrar es la última del mensaje o conversación.

- **public boolean isEsConversacion()**

Devuelve true si la línea que se va a mostrar pertenece a una conversación con un personaje no jugador (y false si, por el contrario, es un mensaje que informa del resultado de una acción).

El segundo constructor de esta clase (**public LineaTiempo(String linea, java.awt.Color c, ClienteJugador cJ, boolean esLaUltimaLinea)** fue eliminado tras comprobar que no se utilizaba.

3.2.5.12. La clase ClienteJugador

Esta clase representa el **MessageListener** sobre el que el usuario realizará todas las interacciones con el mundo virtual de AGM.
En el proceso de reingeniería de esta clase han sido eliminadas cerca de 300 líneas de código correspondientes a variables y métodos que no se utilizaban, conservándose la misma funcionalidad.

Atributos

Fueron eliminadas las siguientes:

- **private String textoConversacion;**

Sólo se utilizaba en los métodos eliminados *setTextoConversacion* y *getTextoConversacion*.

- **private int estadoFrase;**

Esta variable estaba relacionada con la máquina de estados que regía el cambio de la frase del display.

- **private java.util.Map posicionAObjeto**

Esta variable se eliminó porque no se utilizaba.

- **private boolean pedidoCrearPJ;**

Se ponía a true al ejecutar *solicitudCreacionPJ* pero no era consultada desde ningún sitio.

- **private boolean pedidaConexionPJ;**

Esta variable se eliminó porque no se utilizaba.

- **private Collection inventFijo;**
- **public boolean esInventarioFijo(String nombre)**

La eliminación de estas variable tiene su origen en la forma en que está concebido el inventario en la nueva interfaz.

- **private boolean respuestaRecibida;**

Se utilizaba en el método eliminado *responder*.

- **public ObjetosRefresco objetos = null;**

Esta variable se eliminó porque no se utilizaba (ya está objetosRefresco).

De este modo los atributos existentes en la nueva InterfazGrafica son:

- **public PersonajeJugador pJ;**
  Objeto de la propia clase.

- **private static InterfazGrafica iG;**
  Objeto de la clase **IntefazGrafica** (que implementa la GUI de la aplicación).

- **private java.util.List objetosEnPantalla;**
  Lista de los objetos, personajes no jugadores y jugadores existentes en la habitación donde se encuentra el jugador.

- **private HabitacionHome habitacionHome;**

Representa el interfaz home de la clase **Habitación.**

- **private Habitacion habitacion;**

Representa la habitación en la que se encuentra el jugador. Se obtiene utilizando el método *findByPrimaryKey* de **HabitacionHome**.

- **private InitialContext ctx;**

Representa el contexto inicial, que será válido mientras lo sea la sesión del jugador.

- **private QueueConnection qC;**

Representa la conexión a COLA_KERNEL para el envío de las peticiones de creación, conexión y desconexión.

- **private TopicConnection tC;**

Representa la conexión a los topic/idHab (para recibir las notificaciones sobre los cambios de estado en la habitación) y topic/CONFIRMACION (para obtener las respuestas a las solicitudes de creación y conexión).

- **private javax.jms.Queue colaKernel;**

Representa la cola por la que el ClienteJugador envía al sistema las solicitudes de creación del personaje, conexión y desconexión.

- **private TopicSubscriber subscriptor;**

Representa el topic al que está suscrito el jugador (topic/idHab o topic/CONFIRMACION).

- **public String objeto1IdFrase;**
- **public String objeto2IdFrase;**
- **public String idAccion;**

Representan la acción que pretende realizar el jugador.

- **private String fondo;**

Representa el nombre de la imagen de la habitación donde se encuentra el jugador.

- **private String idPNJBloqueante;**

Si se está en medio de una conversación, representa el nombre del personaje no jugador con el que se está hablando y se utiliza para que no se pueda procesar una desconexión.

- **private int pixelsAltoPosicionables;**
- **private int pixelsAnchoPosicionables;**
- **private int posXRelativa;**
- **private int posYRelativa;**
- **private int posXRectRef;**
- **private int posYRectRef;**

Atributos relacionados con el cálculo, según la máscara, de la posible posición (X,Y) del jugador.

- **private boolean mostrarBotonesOpciones;**

Valdrá true cuando haya que responder en las conversaciones con un personaje no jugador.

- **public ObjetosRefresco objetosRefresco = null;**

Se utiliza para refrescar la interfaz grafica del jugador tras producirse un cambio de estado.

- **public String idPJ = null;**

Representa el id del jugador.

- **private LinkedList opciones;**

Representa las opciones de respuesta posibles cuando se está conversando con un personaje no jugador.

- **public static final String rutaImagenes = "src/agm/clienteJugador/interfazGrafica/imagenes/";**

Representa la ruta de la carpeta donde se encuentran todas las imágenes utilizadas en la aplicación.

- **private Logger log = Logger.getLogger(this.getClass());**

Representa el objeto con el cual se escriben las excepciones.

- **private PruebaCliente pc;**

Esta variable se utiliza para la integración con el paquete pruebasxml.

Métodos

De la versión original han sido eliminados los siguientes:

- **public String getTextoConversacion()**
- **public void setTextoConversacion(String string)**

Estos métodos accedían a una variable de tipo **JTextArea** (textoConversacion) que no se utilizaba.

- **public String getIdAccion()**
- **public String getObjeto1IdFrase()**
- **public String getObjeto2IdFrase()**

Estos métodos para obtener los atributos idAccion, objeto1IdFrase y objeto1IdFrase no se utilizaban.

- **public int getEstadoFrase()**
- **public void setEstadoFrase(int i)**

La supresión de estos métodos, así como de la variable de tipo **int** estadoFrase, obedece a la nueva forma en que se gestiona el display con la nueva interfaz.

- **public void actualizaEstadoObjeto(String objetoId, String estado)**

Debía ser una versión anterior de *procesaCambioEstado*.

- **public void enviarSMS(String idPJDestino, String texto)**

Debía ser una versión anterior de *envioChat*. La supresión de este método hizo innecesario importar la clase javax.jms.TextMessage.

- **public void solicitudCreacionPJ(DatosCreacion d)**

Debía ser una versión anterior de *solicitudCreacion*.

- **public void solicitudConexionPJ(DatosConexion d)**

Debía ser una versión anterior de *solicitudConexion.*

- **public void escribirTexto(String texto, int segundos)**

Debía ser fruto de una versión (muy) anterior de *escribeConversacion*. Su código era:

```
public void escribirTexto(String texto, int segundos) {
          iG.escribirTexto(texto);
}
```

- **private void unsubscribeDeTopic(String topicString)**
- **private void subscribeTopicHabitacion()**

Debían ser una versión anterior de *subscribeA* (que valdrá para los dos topic: HABITACION y CONFIRMACION).

- **getGestionPersonajeJugadorHome()**

Este método tampoco se utilizaba, su código era el siguiente:

```
    private agm.objetos.personajesJugadores.GestionPersonajeJugadorHome
getGestionPersonajeJugadorHome()
    throws NamingException {
            return (agm.objetos.personajesJugadores.GestionPersonajeJugadorHome)
getContext().lookup(
            agm.objetos.personajesJugadores.GestionPersonajeJugadorHome.JNDI_NAME);
    }
```

- **public void responder()**

No se utilizaba. Su código era el siguiente:

```
    public void responder() {
            respuestaRecibida = true;
    }
```

De este modo, los métodos del nuevo **ClienteJugador** son:

- **public ClienteJugador()**

Constructor de la clase, inicializa las conexiones JMS

- **public ClienteJugador(PruebaCliente pc)**

Constructor cuando se están ejecutando las pruebas XML.

- **private static String cargarIp()**

Método llamado por *getContext* para obtener el contexto inicial. Carga la IP del servidor de un fichero XML.

- **private static InitialContext getContext() throws NamingException**
- **private HabitacionHome getHabitacionHome() throws NamingException**

Métodos para obtener el contexto inicial y la HabitacionHome a la que está conectado el jugador.

- **private TopicSubscriber subscribeA(String topicString, String messageSelector)**
- **public TopicSubscriber subscribeA(String topicString, String messageSelector, MessageListener ml)**

Métodos para suscribirse a los dos topic: HABITACION (cola por la que se le envía al jugador el resultado de ejecutar alguna acción sobre **HabitacionBean**) y CONFIRMACION (cola por la que se le envía al jugador el resultado de ejecutar alguna acción sobre **OyenteBean**, esto ocurre al crear, conectar o desconectar).

- **public TopicSubscriber getSubscriptor()**
- **public void setSubscriptor(TopicSubscriber subscriber)**

Métodos para acceder a la variable subscriptor (de tipo **TopicSubscriber**).

- **private QueueConnection creaQueueConnection()**

Se le llama desde el constructor de la clase. Crea la conexión a COLA_KERNEL para el envío de las peticiones de creación, conexión y desconexión.

- **private TopicConnection creaTopicConnection()**

Se le llama desde el constructor de la clase. Crea las conexiones a topic/idHab (para recibir las notificaciones sobre los cambios de estado en la habitación) y topic/CONFIRMACION (para obtener las respuestas a la solicitudes de creación y conexión).

- **public static void main(String[] args)**

Es el arranque de un jugador que consiste en mostrar el panel de bienvenida mediante un **JOptionPane** el cual dará lugar a la creación de un personaje o a la conexión de uno que ya estuviera creado.

- **public void solicitudCreacionPJ(DatosCreacion d, MessageListener ml)**

Método invocado desde el *main*. Se crea un mensaje de SOLICITUD_CREACION_PJ, encapsulando el id, el nombre y el password del jugador, y se envía a la cola CONFIRMACION.

- **public void solicitudCreacionPJ()**

Para las pruebas XML.

- **public void solicitudConexionPJ(DatosConexion d, MessageListener ml)**

Método invocado desde el *main*. Se crea un mensaje de SOLICITUD_CONEXION_PJ, encapsulando el id y el password del jugador, y se envía a la cola CONFIRMACION.

- **public void solicitudConexionPJ()**

Para las pruebas XML.

- **private void cierraClienteJugador(int codigo)**

Se cierra **ClienteJugador** y para ello cierra todos sus recursos JMS (QueueConnection y TopicConnection). Se le llama si no tiene éxito una solicitud de creación o de conexión, si se cierra mediante el aspa los paneles de bienvenida, creación o conexión o si se intenta ejecutar **ClienteJugador** antes de **ClienteAdministrador**.

- **public void solicitudDesconexionPJ()**

Método invocado al pulsar el botón Desconectar o hacer click en el aspa de la interfaz. Se crea un mensaje de SOLICITUD_DESCONEXION_PJ, encapsulando el id y el password del jugador, y se envía a la cola CONFIRMACION.

- **private void escribeConversacion(LinkedList conversacionFija, boolean empiezaJugador, String nombrePNJ)**

Invoca al método *escribirConversacion* de **InterfazGrafica** con la lista resultante de concatenar los fragmentos de la conversación que recibe como parámetro (tras darles un nuevo formato de color, etc.).

- **public void mensajeMostrado()**

Se encarga de ocultar la caja de opciones y mostrar los botones Ejecutar, Enviar y Desconectar (o viceversa) en función de si el mensaje forma parte de una conversación fija o no.

- **private void interpretaMascara(String masc)**

Calcula según la máscara la posible posición (X,Y) del jugador.

- **private void activaDebug()**

Activa el nivel de debug para el logging.

- **public void onMessage(Message message)**

Recibe el message y delega en el método *procesa*.

- **private void procesa(javax.jms.Message message)**

Invocado desde el método *onMessage*, discrimina sobre lo distintos tipos de mensaje (CONFIRMACION_CREACION_PJ, CONFIRMACION_CONEXION_PJ, CONFIRMACION_DESCONEXION_PJ, DENEGACION_CONEXION, MENSAJE_CHAT, NOTIFICACION_TEXTUAL, OBJETO_COMUNICACION y OBJETOS_REFRESCO) y llama a su correspondiente método para procesarlo.

- **public void muestraPantalla(javax.jms.Message message)**

Si el método *procesa* recibe un message de tipo CONFIRMACION_CONEXION_PJ se llama a este método el cual suscribe al jugador a su cola correspondiente y da una situación en pantalla al jugador.

- **public void refresco(javax.jms.Message message)**

Construye un objeto de tipo **ObjetoRefresco** a partir del message que recibe y lo envía a *procesaObjeto*.

- **public void notificacion(javax.jms.Message message)**

Construye un objeto de tipo **NotificacionTextual** a partir del message que recibe, discrimina sobre los distintos tipos de notificaciones textuales (**SacaPJ**, **MetePJ**, **SacaInv**, **MeteInv** y **CambioEstado**) y lo envía a su correspondiente método.

- **public void comunicacion(javax.jms.Message message)**

Construye un objeto de tipo **ObjetoComunicacion** a partir del message que recibe y lo envía a *procesaObjeto*.

- **public void procesaObjeto(ObjetosRefresco oR)**

Recibe un objeto de tipo **ObjetoRefresco** (objetos+personajes+jugadores) y refresca toda la **InterfazGrafica**.

- **private void procesaObjeto(ObjetoActualizacion oAC)**

Recibe un objeto de tipo **ObjetoActualizacion**, itera sobre la lista de notificaciones que lo forman y procesa cada una de ellas por separado.

- **public void procesaNotificacionTextual(NotificacionTextual nT)**

Obtiene el texto a mostrar del objeto de tipo **NotificacionTextual** que recibe como parámetro y llama a *escribirTexto* de **InterfazGrafica**.

- **public void procesaObjeto(ObjetoComunicacion oC)**

Recibe un objeto de tipo **ObjetoComunicacion** (conversación fija+opciones de respuesta), discrimina si pertenece a una conversación fija y delega hasta mostrar el mensaje en la consola de la interfaz.

- **public void procesaSacaPJ(SacaPJ sPJ)**

Recibe un objeto de tipo **SacaPJ** (jugador+mensaje). El jugador lo elimina de la lista de objetos en pantalla de la habitación. Llama a *escribirTexto* de **InterfazGrafica** pasándole como parámetro el mensaje.

- **public void procesaMetePJ(MetePJ mPJ)**

Recibe un objeto de tipo **MetePJ** (jugador+mensaje). El jugador lo añade a la lista de objetos en pantalla de la habitación (da al jugador una situación, aleatoria, en pantalla). Llama a *escribirTexto* de **InterfazGrafica** pasándole como parámetro el mensaje.

- **public void procesaSacarInv(SacarInv sI)**

Recibe un objeto de tipo **SacarInv** (objeto+mensaje). El objeto lo borra de la colección de objetos del jugador. Llama a *escribirTexto* de **InterfazGrafica** pasándole como parámetro el mensaje. Y refresca el inventario.

- **public void procesaMeterInv(MeterInv mI)**

Recibe un objeto de tipo **MeterInv** (objeto+mensaje). El objeto lo añade a la colección de objetos del jugador. Llama a *escribirTexto* de **InterfazGrafica** pasándole como parámetro el mensaje. Y refresca el inventario.

- **public void procesaCambioEstado(CambioEstado cE)**

Procesa el cambio de estado de un objeto que tiene el jugador. Recibe un objeto de tipo **CambioEstado** (objeto+estado+mensaje). Se llama al método cambiaEstado de dicho objeto. Llama a *escribirTexto* de **InterfazGrafica** pasándole como parámetro el mensaje. Y refresca el inventario.

- **public List creaObjetosEnPantalla(java.util.List objetosNoPersonajes, java.util.List personajesNJugadores, java.util.List personajesJugadores)**

Este método devuelve la lista de objetos en pantalla resultante de iterar sobre las tres listas que recibe como parámetro (de objetos, personajes jugadores y personajes no jugadores).

- **public List getObjetosEnPantalla()**

Se invoca desde la clase **Pantalla** para obtener la lista de los objetos que hay en la pantalla (habitación) donde se encuentra el jugador.

- **private ObjetoEnPantalla buscarObjetoEnPantalla(String idO)**

Devuelve el objeto en pantalla cuyo nombre recibe como parámetro.

- **private PersonajeJugador buscarPersonaje(List lista, String idPJ)**

Itera en la lista y devuelve el personaje jugador cuyo id recibe como parámetro.

- **public void ejecutaAccion()**

Se obtiene la habitación en que se encuentra el jugador y, tras comprobar que la acción que pretende llevar a cabo el jugador tiene sentido, se delega en el método *ejecutaAccion* de **HabitacionBean**.

- **public void teRespondo(String idOpcion)**

Se le llama desde **OyenteOpcion** de **InterfazGrafica** cuando se elige una respuesta al conversar con un personaje no jugador. Obtiene la habitación en que se encuentra el jugador y llama a *teRespondo* de **HabitacionBean** con la opción elegida, el id del jugador y el id del personaje (bloqueante).

- **public void envioChat(String msg)**

Se le llama desde **OyenteEnvioMsgChat** de **InterfazGrafica**. Obtiene la habitación en que se encuentra el jugador y llama a *envioChat* de **HabitacionBean** con dicho mensaje y el nombre del jugador (que será lo que se muestre en el panel entrante del chat).

- **private boolean esPNJ(String idPNJ)**

Devuelve true si el id que recibe como parámetro corresponde a un personaje no jugador.

- **private boolean esPJ(String idPJ)**

Devuelve true si el id que recibe como parámetro corresponde a un jugador.

- **public String getIdPNJBloqueante()**

Se le llama desde **OyenteDesconectar** de **InterfazGrafica** de tal forma que si hay algún PNJBloqueante (esto es, si se está conversando con un personaje no jugador) no se lleve a cabo la solicitud de desconexión.

- **public void setIdAccion(String accion)**
- **public void setObjeto1IdFrase(String objeto1)**
- **public void setObjeto2IdFrase(String objeto2)**

Se invocan desde **OyenteEjecutar** de **InterfazGrafica** para actualizar los atributos idAccion, objeto1IdFrase y objeto2IdFrase de **ClienteJugador**.

- **public String getFondo()**

Se le llama desde la clase Pantalla para obtener el nombre de la imagen de fondo (la habitación) donde se encuentra el jugador.

- **public PersonajeJugador getPJ()**

Devuelve el **PersonajeJugador** asociado a **ClienteJugador**.

3.2.5.13. La clase Teclado

Permite la lectura de datos desde la entrada estándar. Es decir, trata de hacer transparente la lectura de los tipos más comunes de datos desde la entrada estándar (por ejemplo, sin tener que tener en cuenta excepciones ya que son capturadas dentro de la propia clase, y sin tener que introducir las clases de entrada/salida -streams-).
Se necesita para poder introducir desde teclado los datos para crear un personaje y conectarlo y para poder conversar con otros jugadores en el chat.

Atributos

- **public static BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in))**

Variable de clase asignada a la entrada estándar del sistema.

Métodos

- **public static String leerString()**

Lee una cadena desde la entrada estándar.

- **public static int leerInt()**

Lee un número entero desde la entrada estándar.

- **public static float leerFloat()**

Lee un numero real (float) desde la entrada estándar.

- **public static double leerDouble()**

Lee un numero real (double) desde la entrada estándar.

- **public static char leerChar()**

Lee un carácter desde la entrada estándar.

3.2.6. Análisis de componentes

3.2.6.1. Paneles de bienvenida, creación y conexión

La independencia de estos tres paneles de la pantalla de juego propiamente dicha los convirtió en un escenario ideal para familiarizarse con las APIs AWT y SWING.

En este punto, algunas de las páginas consultadas en Internet fueron las siguientes:

Tutorial de AWT y SWING: www.itapizaco.edu.mx
Tutorial de SWING: www.programacion.net
JOptionPane: ji.ehu.es
La clase Box y la distribución: ww.programacion.com
Layout managers: www.usuario.com
Layout de un contenedor: www.dcc.uchile.cl

Sobre las APIs elegidas, cabe señalar que, desechada SWT (más rápida pero menos portable), y ante la posibilidad de convertir completamente la aplicación a Swing o seguir utilizando AWT/Swing, se optó por esto último utilizando prioritariamente AWT para la gestión de eventos y Swing para los componentes (algo más sofisticados) a la espera de que Swing se fusione más profundamente con AWT en futuras versiones de Java (www.javahispano.org, www.clubdevelopers.com, etc.).

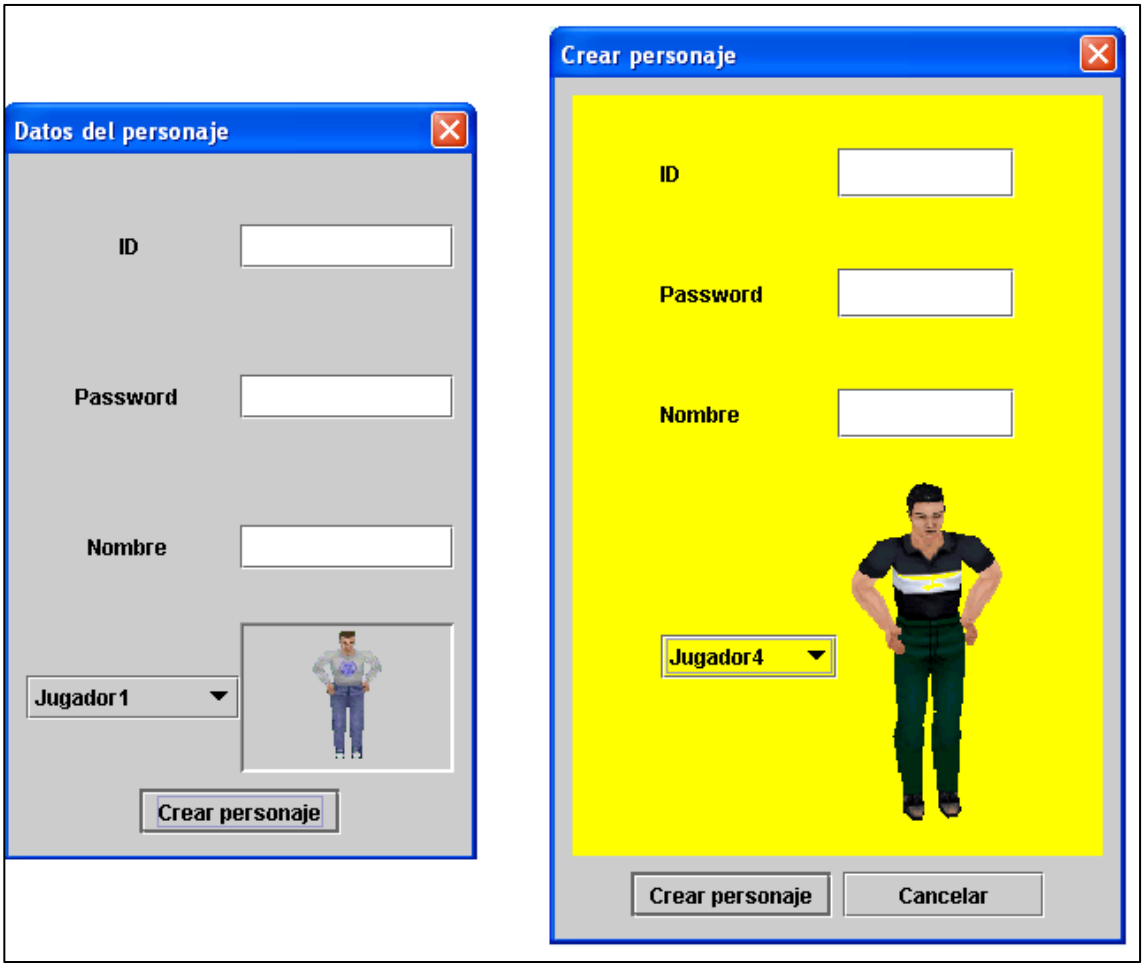
Respecto a los tres paneles se comprobó que el de creación del personaje y el de conexión eran similares: ambos se implementaban en una clase del paquete agm.clienteJugador.interfazGrafica, extendían la clase JPanel y eran añadido a otro de tipo JOptionPane (un soporte para mostrar diálogos estándares) en la clase ClienteJugador. El de bienvenida simplemente era un panel de tipo JOptionPane.

A continuación se estudian el aspecto original, las mejoras gráficas obtenidas para cada uno de ellos, así como los cambios más significativos en el código (que ha quedado completamente comentado).

Panel de Creación

Comparativa visual

Fue el primero con el que se empezó a trastear. La comparativa entre el aspecto original y el actual es la siguiente:



3.7. Cambios en el panel de creación

Cambios a nivel visual

Se añade el botón cancelar, se agranda la imagen de los posibles jugadores, se añade un color de fondo y se amplía la separación entre componentes.

Cambios a nivel de código

El panel original era la consecuencia del siguiente código en el método solicitudCreacionPJ() de ClienteJugador.java:

```
PanelCreacion panelCreacion = new PanelCreacion();
Object[] texto2 = { "Crear personaje" };
if (JOptionPane.showOptionDialog(null, panelCreacion, "Datos del personaje",
        JOptionPane.OK_OPTION,JOptionPane.PLAIN_MESSAGE,null,texto2,texto2[0])
        == JOptionPane.OK_OPTION)
```

Así pues, el panel donde se solicitan los datos para crear el personaje era el resultado de añadir un JPanel a un JOptionPane (siendo éste el responsable de añadir el botón “Crear personaje”).

En cuanto al JOptionPane se cambió por uno de tipo YES_NO_OPTION para añadir el botón Cancelar.

En lo referente a la clase PanelCreacion la distribución anterior consistía en un GridLayout(4,2) en el que se iban añadiendo cada uno de los ocho componentes: los tres JLabel, los tres JTextField, el JComboBox y un JButton con la imagen asociada.

Y la distribución fue cambiada a la siguiente: por un lado un GridLayout(3,2) para los tres JLabel y los tres JTextField. Por otro lado un GridLayout(1,2) para el JComboBox y el JLabel con la imagen asociada (en lugar del JButton que había antes).

Ambos paneles serían incluidos en un GridLayout(2,1) el cual sería incluido en el centro de un BorderLayout final.

Todo esto puede parecer algo engorroso pero los Hgap y Vgap de GridLayout no respondían correctamente y sin ellos era demasiado rígido. Era necesaria esta combinación de paneles y controladores para lograr el resultado final, con los tamaños de los componentes y los huecos entre sí deseados.

Cabe señalar que se intentó quitar infructuosamente el borde del JComboBox (posible bug de Sun) y se estableció el color de fondo como constante para facilitar su posible cambio.

Panel de Conexión

Comparativa visual

La comparativa entre el aspecto original y el actual del panel de conexión es la siguiente:

3.8. Cambios en el panel de conexión

Cambios a nivel visual

Se añade el botón cancelar, la imagen de las llaves, un color de fondo y se amplía la separación entre componentes.

Cambios a nivel de código

El panel original era la consecuencia del siguiente código en el método solicitudConexionPJ() de ClienteJugador.java:

```
PanelLogin panelLogin = new PanelLogin();
Object[] texto1 = { "Conectar" };
if (JOptionPane.showOptionDialog(null, panelLogin, "Conexión",
      JOptionPane.OK_OPTION,JOptionPane.PLAIN_MESSAGE,null,texto1,texto1[0])
       == JOptionPane.OK_OPTION)
```

Del mismo modo que el de creación, el panel donde se solicitan los datos para conectar el personaje era el resultado de añadir un JPanel a un JOptionPane (siendo éste el responsable de añadir el botón “Conectar personaje”).

Al igual que en el caso anterior se cambió el JOptionPane por uno de tipo YES_NO_OPTION para añadir el botón Cancelar.

En lo referente a la clase PanelLogin la distribución anterior consistía en un GridLayout(2,2) en el que se iban añadiendo cada uno de los cuatro componentes: los dos JLabel y los dos JTextField.

Y la distribución fue cambiada a la siguiente: por un lado un GridLayout(2,2) para los dos JLabel y los dos JTextField. Por otro lado un BorderLayout en cuyo oeste sería incluida el JLabel con la imagen y en cuyo centro sería incluido el GridLayout.

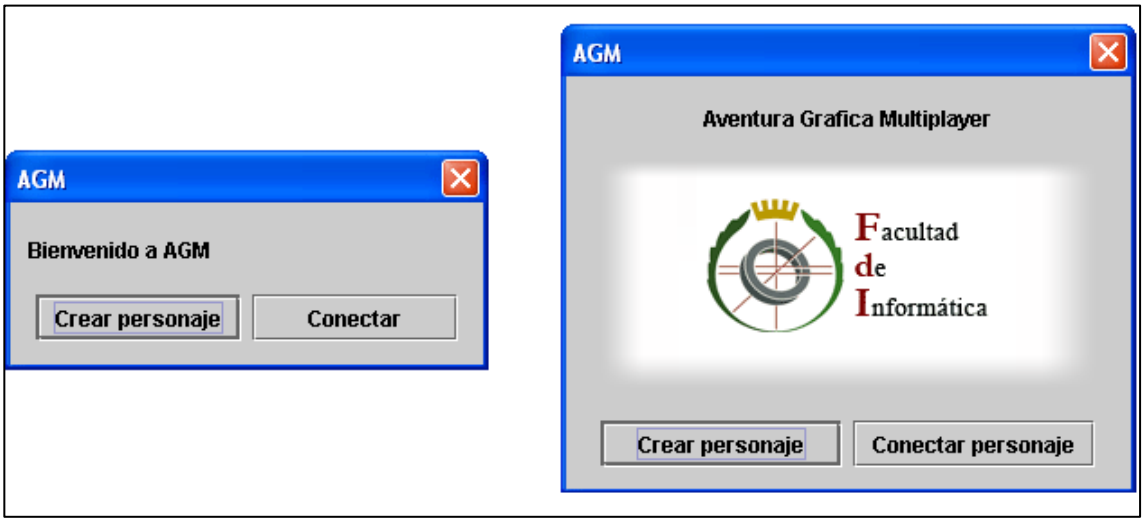
Del mismo modo que en el panel anterior, esto puede parecer algo engorroso pero los Hgap y Vgap de GridLayout no respondían correctamente y sin ellos era demasiado rígido. Era necesaria esta combinación de paneles y controladores para lograr el resultado final, con los tamaños de los componentes y los huecos entre sí deseados.

Al igual que en el caso anterior el color de fondo fue establecido como constante para facilitar su posible cambio.

Panel de Bienvenida

Comparativa visual.

La comparativa entre el aspecto original y el actual del panel de bienvenida es la siguiente:



3.9. Cambios en el panel de bienvenida

Cambios a nivel visual.

Se centra el título y se añade una imagen de bienvenida.

Cambios a nivel de código.

El aspecto original era la consecuencia del siguiente código en el main de ClienteJugador.java:

```
int opcion = JOptionPane.showOptionDialog(null, "Bienvenido a AGM",
             "AGM", JOptionPane.YES_NO_OPTION,
             JOptionPane.PLAIN_MESSAGE, null, texto, texto[0]);
```

Buscando la uniformidad con los paneles anteriores (y una bienvenida más vistosa) se añade al paquete la clase PanelBienvenida y se cambia el trozo de código anterior por el siguiente:

```
PanelBienvenida panelBienvenida = new PanelBienvenida();
            Object[] texto = { "Crear personaje", "Conectar personaje" };
            int opcion = JOptionPane.showOptionDialog(null, panelBienvenida,
                   "AGM", JOptionPane.YES_NO_OPTION,
                   JOptionPane.PLAIN_MESSAGE, null, texto, texto[0]);
```

El panel de bienvenida es simple: un JPanel con distribución BorderLayout en cuyo centro se ha incluido un JLabel con el texto y la imagen de presentación.

En este punto se plantea la conveniencia de utilizar JFrame para cada uno de estos tres paneles en lugar de insertar un JPanel en un JOptionPane, de manera que se

obtuvieran resultados menos rígidos y más vistosos, reservando los JOptionPane para mensajes de tipo confirmación o error.

De esta forma podrían tenerse paneles de bienvenida como el siguiente (boceto):

3.10. Boceto de un panel de bienvenida con JFrame

3.2.6.2. Pantalla de juego y consola

Comparativa visual

3.11. Cambios en la pantalla de juego

Cambios a nivel visual

A nivel visual se ha ajustado la imagen de fondo al tamaño máximo del panel que la contiene (antes se veía el panel) y se han eliminado el divisor del JSplitPane y un rectángulo que aparecía en la parte inferior (que debía estar relacionado con pruebas de la animación). Se ha agrandado la imagen en altura y se ha ubicado en las dos terceras partes superiores de la pantalla tal y como aparecía en la versión MS-DOS.

Estos cambios en el tamaño y ubicación de la pantalla de juego han hecho necesario que se reconsideraran las coordenadas de los objetos y personajes (que ya ofrecían errores de cálculo en la versión original). Dicha modificación fue llevada a cabo en la clase **GestorSistemaBean**.

En este punto se aconseja tomar nuevas fotografías de las distintas habitaciones, vistas de frente y con una correlación en cuanto al tamaño para que la interfaz gráfica quede mejor.

Cambios a nivel de código

Inicialmente la pantalla de juego estaba concebida como un JSplitPane que incluía dos componentes: la pantalla propiamente dicha y la consola.

Esto sigue siendo así, pero como se puede apreciar en los árboles de la introducción se han eliminado hasta cuatro componentes que los embutían y que no hacían sino oscurecer el código (el cual ha sido reducido a una tercera parte).

De este modo la pantalla (antes de tipo "**Bot**") deja de estar incluida en un **JScrollPane** y a su vez en un **Box** y a su vez en otro **Box** para (ahora de tipo "**Pantalla**") estar contenida directamente en el **JSplitPane**.

La clase **Pantalla** es necesaria ya que para insertar una imagen de fondo en un panel hay que crear una clase que herede de **JPanel** y sobreescribir su método *paintComponent* (ver enlace).

La línea *g.drawString(cJ.getTextoConversacion(), 100, 550);* del método *paint* de la clase **Pantalla** sobra (posiblemente se trate de una prueba antes de concebir la consola) y el método *getTextoConversacion* sobra también puesto que siempre devuelve "".

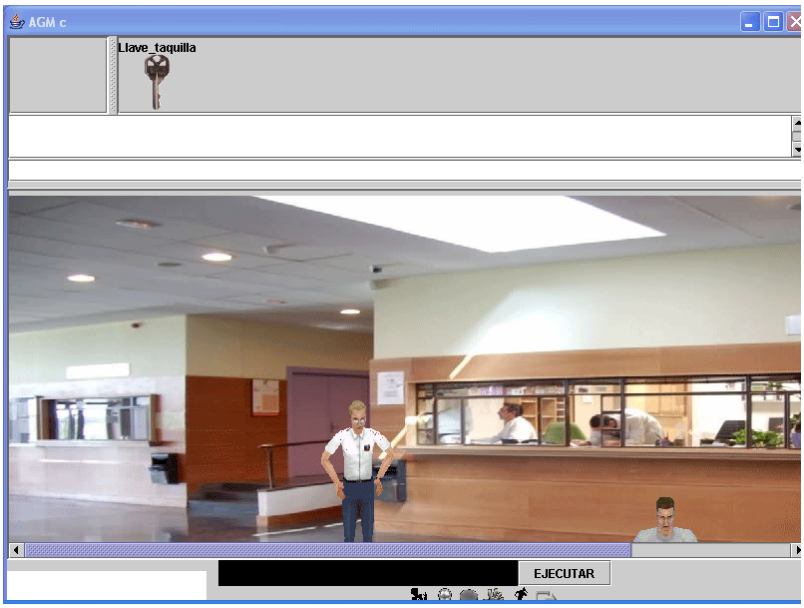
Habría que estudiar con más detenimiento el método *solapa*, que devuelve true si se solapan dos objetos de la pantalla. Parece no tener sentido si se van pintando unos sobre otros.

Las clases **Celda** y **OyenteCelda**, tras comprobar que no realizan ninguna función y que tampoco estaban relacionadas con la versión animada, son eliminadas (para obtener las coordenadas de los objetos o personajes presentes en la pantalla se utiliza *getX* y *getY*).

La consola sigue siendo un **JTextArea** incluido en un **JScrollPane** para aquellos mensajes demasiado largos aunque está pendiente una solución mejor.

Los métodos *ocultaConsolaNoEditable* y *muestraConsolaNoEditable* presentes en **InterfazGrafica** sobran puesto que esto ya se hace en **EscritorTexto**.

Cabe señalar que el incluir la pantalla en un **JScrollPane** era un artificio para tratar de implementar el cambio de configuración. Así, para una resolución de 800x600 quedaba de la siguiente forma:



3.12. Aspecto de la interfaz original para una resolución de 800x600

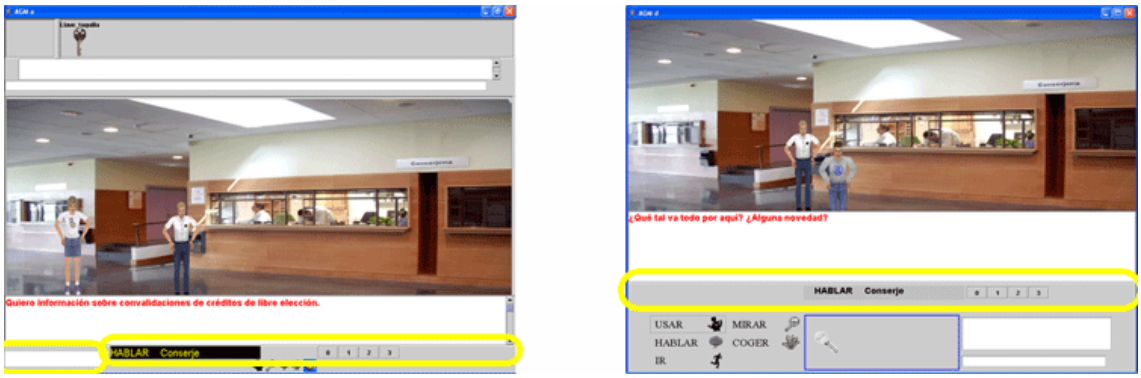
Aunque en esta versión del juego no está implementado el cambio de configuración queda claro que la salida no es un **JScrollPane** sino escalar la imagen (ver apartado “Trabajo futuro”).

Como se comentaba antes, el código concerniente a estos dos componentes de la interfaz (pantalla de juego y consola) se ha comentado adecuadamente y ha sido reducido a una tercera parte.

En torno a la pantalla de juego y la consola se podrían implementar una serie de mejoras las cuales han sido apuntadas en el apartado “Trabajo futuro”.

3.2.6.3. Display

Comparativa visual



3.13. Evolución visual del display

Cambios a nivel visual

Se ha prescindido del JTextArea objetoSeleccionado, reservando el display únicamente a la frase que representa la acción que pretende realizar el jugador.

Siguiendo el ejemplo de la versión MS-DOS, se ha colocado entre la pantalla de juego y los componentes del panel inferior, se ha centrado y se le ha cambiado el color de fondo para que todo quede más uniforme.

Cambios a nivel de código

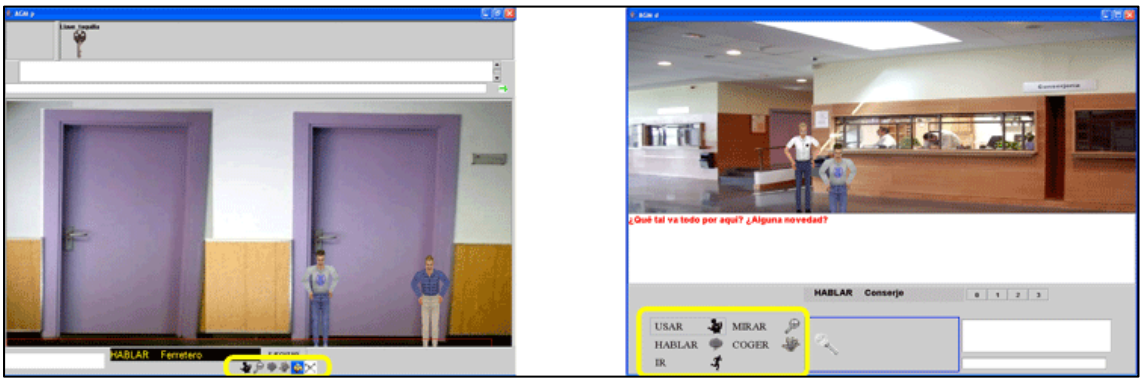
Los cambios a nivel de código relacionados con el display ya se comentan detalladamente al analizar la clase InterfazGrafica. Simplemente recordar que para prescindir del JTextArea objetoSeleccionado hubo que realizar numerosas modificaciones en los oyentes de la pantalla y del inventario y la forma en que se relacionaban con el display.

Una posible mejora relacionada con el display se apunta en el apartado “Trabajo futuro”.

3.2.6.4. Caja de acciones

Comparativa visual

La diferencia entre la botonera de acciones antes y después puede apreciarse en la siguiente *snapshot*:



3.14. Comparativa visual de la caja de acciones

A continuación se muestra la botonera de acciones con mayor detalle:

3.15. Comparativa visual de la caja de acciones (detalle)

Cambios a nivel visual

A nivel visual, y siguiendo el ejemplo de la versión MS-DOS, la botonera de acciones se amplía y se sitúa en la esquina inferior izquierda de la pantalla.

Las imágenes se agrandan y, para algunas acciones, se buscan unas más significativas y acordes con el resto de la interfaz.

En lugar de un *tool tip* se acompaña a las imágenes del nombre de la acción para facilitar la jugabilidad.

Cambios a nivel de código

La botonera de acciones deja de ser un Box para convertirse en un JPanel con una composición GridLayout(2,3), esto es una matriz de dos filas y tres columnas, con una celda para cada uno de los seis botones.

El formato de las imágenes de los botones, que antes era pasado como parámetro al método creaBotonAccion, se unifica a gif (ideal para este tipo de aplicaciones como ya se ha comentado) al tiempo que se amplían de tamaño y se las dota de una mayor resolución.

En cuanto al actionPerformed de OyenteAccion se modifica para implementar el cambio del puntero. Cada vez que se pulse un botón de acción el puntero del ratón deja de ser la tradicional flecha para convertirse en la imagen asociada a dicha acción (también en formato gif).

De este modo el puntero adoptará las siguientes formas (el botón desconectar no se incluye por razones obvias):

3.19 Distintas imágenes para el cambio de puntero

.

Para implementar el cambio de puntero hubo que importar las clases Cursor, Toolkit y Point (todas ellas presentes en el paquete java.awt). Las páginas consultadas fueron las siguientes:

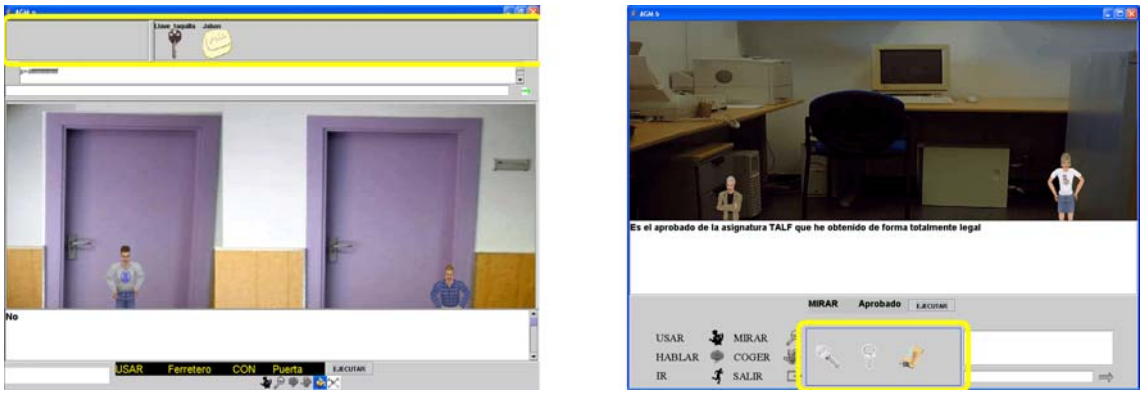
java.sun.com
www.lawebdelprogramador.com
www.koders.com

La visibilidad del nuevo puntero queda reducida sólo al ámbito de la pantalla de juego (no se puede apreciar mediante *snapshots* ya que éstas no capturan el puntero del ratón pero se adjunta vídeo flash).

En el apartado “Trabajo futuro” se han apuntado una serie de mejoras que podrían realizarse en torno a este componente de la interfaz.

3.2.6.5. Inventario

Comparativa visual



3.20. Comparativa visual del inventario

Cambios a nivel visual

A nivel visual no se entendía la utilización de un **JSplitPane**, un tipo de panel que debería utilizarse para visualizar dos componentes, uno a cada lado, con la posibilidad de modificar la cantidad de espacio otorgado a cada uno (zarza.usal.es). Pero su componente izquierdo parecía no tener ninguna función puesto que era únicamente en su componente derecho donde se reflejaba el objeto inicial con que partía el jugador (la llave de la taquilla) así como el resto de objetos que iba metiendo o sacando (la pastilla de jabón, la llave del despacho, etc.).

En cuanto a su ubicación, parecía más lógica su situación en la versión MS-DOS: junto a la botonera de acciones en la parte inferior de la pantalla.

La etiqueta que acompañaba al objeto parecía redundante y, buscando una mayor limpieza de la interfaz, sería sustituida por un *tool tip* que mostraría el nombre del objeto al situarse encima el puntero del ratón.

Cambios a nivel de código

En un principio el inventario estaba concebido como un Box que contenía un JSplitPane que a su vez contenía un JScrollPane. En este JScrollPane se añadía un Box al que a su vez se le iban añadiendo elementos de tipo Box formados por un JLabel y un JButton.

¿Pero por qué se utilizaba un JSplitPane en el que el divisor se podía mover y en el que sólo parecían objetos en su componente derecho? A nivel de código se pudo deducir que el inventario pretendido estaba dividido en dos partes: a la izquierda un inventario fijo y a la derecha uno que se iba refrescando.

Podía ser que en la situación en que recibimos la aplicación sólo se utilizara la parte no-fija del inventario y que se hubiera reservado una parte fija, por razones de eficiencia, para otras misiones. Pero realmente el hecho de meter o sacar un objeto se

produce muy pocas veces a la hora de jugar y por otro lado ningún objeto del inventario parecía más fijo que la propia llave de taquilla.

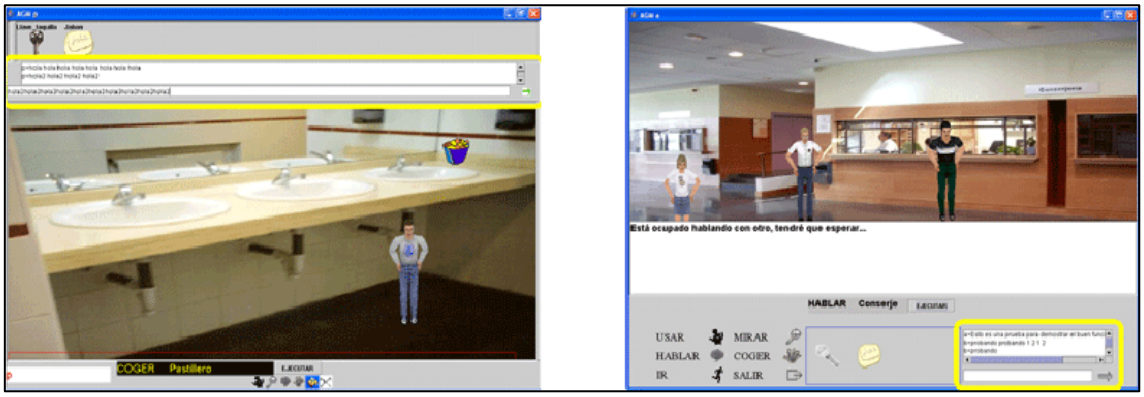
Así pues se decidió sustituir la concepción de un inventario separado en dos partes por un único inventario más sencillo en cuanto a su estructura y más fácil de gestionar, pasando a ser un único JPanel formado por un único Box al que se le añadirían los objetos del jugador.

Como se comenta en el análisis de la clase InterfazGrafica los cinco métodos relacionados con el inventario se redujeron a tres, reduciéndose las líneas de código a un 40% y conservando a efectos visuales la misma funcionalidad.

En el apartado “Trabajo futuro” se apunta una posible mejora consistente en implementar el cambio del puntero al seleccionar un objeto del inventario (adoptando la imagen reducida del objeto) como ocurre al seleccionar las acciones “usar”, “ver”, “hablar”, “coger” o “ir a”.

3.2.6.6. Chat

Comparativa visual



3.21. Evolución visual del chat

Cambios a nivel visual

A nivel visual se añade al panel saliente un *scrollbar* horizontal (antes se perdían los mensajes demasiado largos), que al igual que el *scrollbar* vertical sólo será visible cuando se necesite por el tamaño de los mensajes, se cambia la imagen del botón por una mayor y más acorde con el resto de la interfaz, se centran ambos paneles, se separan un poco más y se sitúan en la esquina inferior derecha de la pantalla, para reservar la parte central y superior a la pantalla de juego.

Cambios a nivel de código

La estructura del componente sigue siendo la misma: un Box que contiene el panel entrante (JTextArea contenida en JScrollPane) y el panel saliente (un Box que contiene un JTextField y un JButton a continuación).

Aparte de la eliminación de todo aquello relacionado con una consola editable de la que se hablaba en la anterior versión (que tenía métodos para mostrarse y ocultarse pero que no se utilizaba en absoluto), los únicos cambios en el código reseñables podrían ser los referentes a los *scrollbars* (ver enlace):

```
JScrollPane aux = new JScrollPane(mensajesEntrantes,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
```

Pasa a ser:

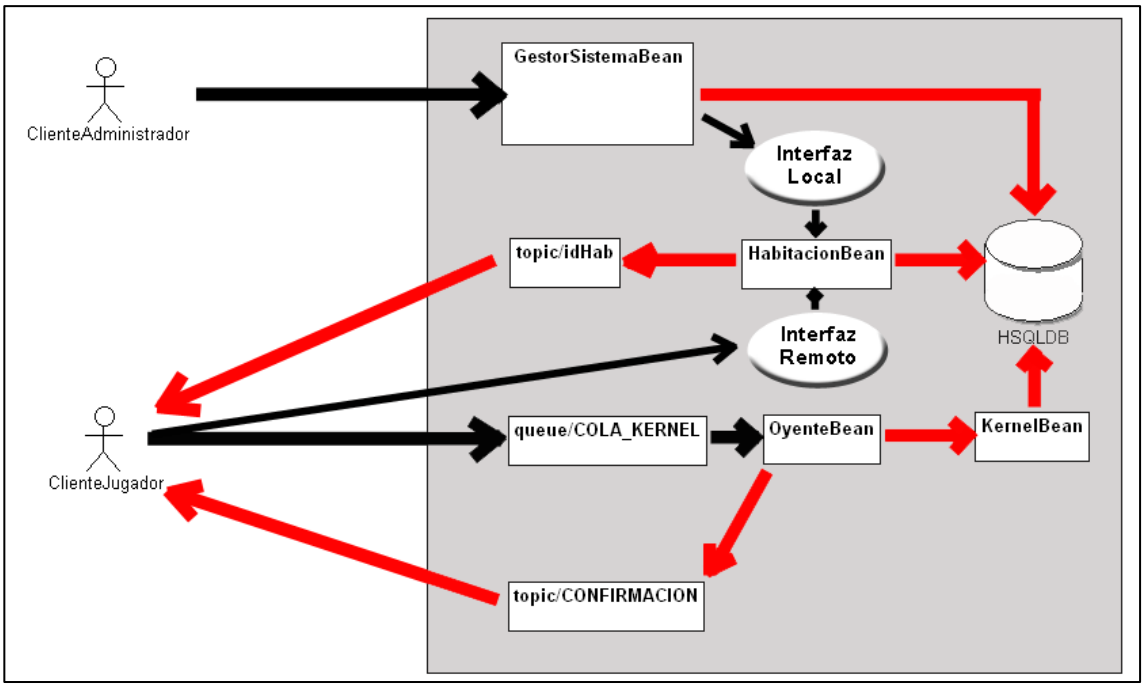
```
JScrollPane panelEntrante = new JScrollPane(mensajesEntrantes,
        JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
        JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
```

Otro cambio que se podría señalar es la utilización de la ruta de las imágenes (en este caso del botón enviar) como constante para facilitar un cambio en el futuro si hubiera lugar.

3.3. Casos de uso

En este apartado estudiaremos la interacción del usuario con la aplicación, por un lado para arrancar el sistema, crear un personaje y conectarlo, y por otro, una vez creada la partida, para ejecutar una acción (ir, coger, usar o mirar), para hablar con un personaje o para hablar con el resto de personajes conectados.

Antes de analizar los diferentes casos de uso conviene analizar la estructura básica con J2EE de nuestra aplicación, que se ha representado en el diagrama siguiente:



3.22. Arquitectura básica con J2EE

En él se pueden distinguir los siguientes elementos:

ARQUITECTURA DEL LADO DEL USUARIO

ClienteAdministrador:
Aplicación que se lanza para solicitar la creación del mundo.

ClienteJugador (MessageDrivenBean)
Aplicación que se lanza para crear y conectar un jugador.

ARQUITECTURA DEL LADO DEL SISTEMA

- **BEANS**

GestorSistemaBean (SessionBean)
Proporciona el servicio de creación del mundo por medio de su interfaz remoto.

KernelBean (SessionBean)
Proporciona el servicio de creación y conexión de jugadores al OyenteBean.

OyenteBean (MessageDrivenBean)
Escucha por Queue/COLA_KERNEL las peticiones de los distintos ClienteJugador para crear o conectar jugadores y las lleva a cabo a través de KernelBean.

HabitacionBean (EntityBean)

Representa cada una de las habitaciones. Escucha peticiones de los jugadores que hay en esa habitación a través de su interfaz remoto y les envía notificaciones por medio de la cola topic/idHab.

- **QUEUE**

  Queue/COLA_KERNEL
  Cola por la que OyenteBean recibe las peticiones de creación y conexión de un ClienteJugador.

- **TOPICS**

  Topic/CONFIRMACION
  Usado por OyenteBean para enviar las respuestas a la solicitudes de creación y conexión.

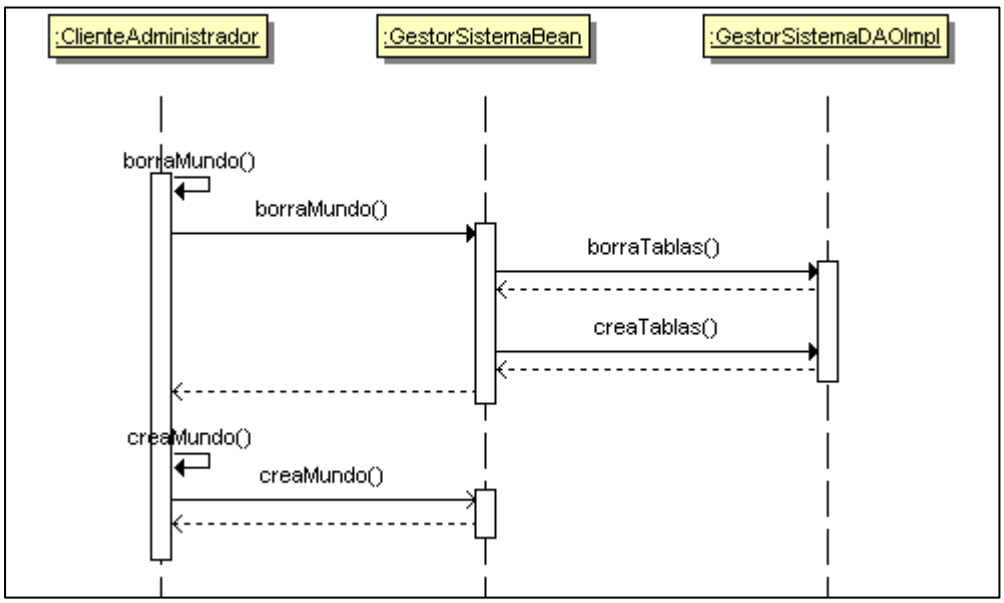
  Topic/idHab
  Usado por HabitacionBean para enviar notificaciones sobre cambios de estado a los ClienteJugador que se encuentren en esa habitación.

El gestor de bases de datos utilizado para gestionar la persistencia es **HSQLDB** (Hypersonic SQL Data Base).

Una vez visto esto se está en mejores condiciones de analizar los diferentes casos de uso. La explicación que se ofrece mediante diagramas de secuencia y flujos de sucesos pretende ser una mejora de la incluida en la documentación del curso 2003/04. Todos los diagramas de secuencia y flujos de sucesos han sido revisados, corregidos y completados con nueva información. Los del apartado 3.3.5. (Hablar con un personaje) se han incluido por primera vez ya que no figuraban en dicha documentación.

3.3.1. Arrancar la aplicación

Diagrama de secuencia



Flujo de sucesos

ClienteAdministrador se encarga de inicializar el mundo del juego utilizando el interfaz remoto **GestorSistemaBean** (de tipo session).

- Para ello ejecuta una llamada *borraMundo()* sobre GestorSistemaBean.

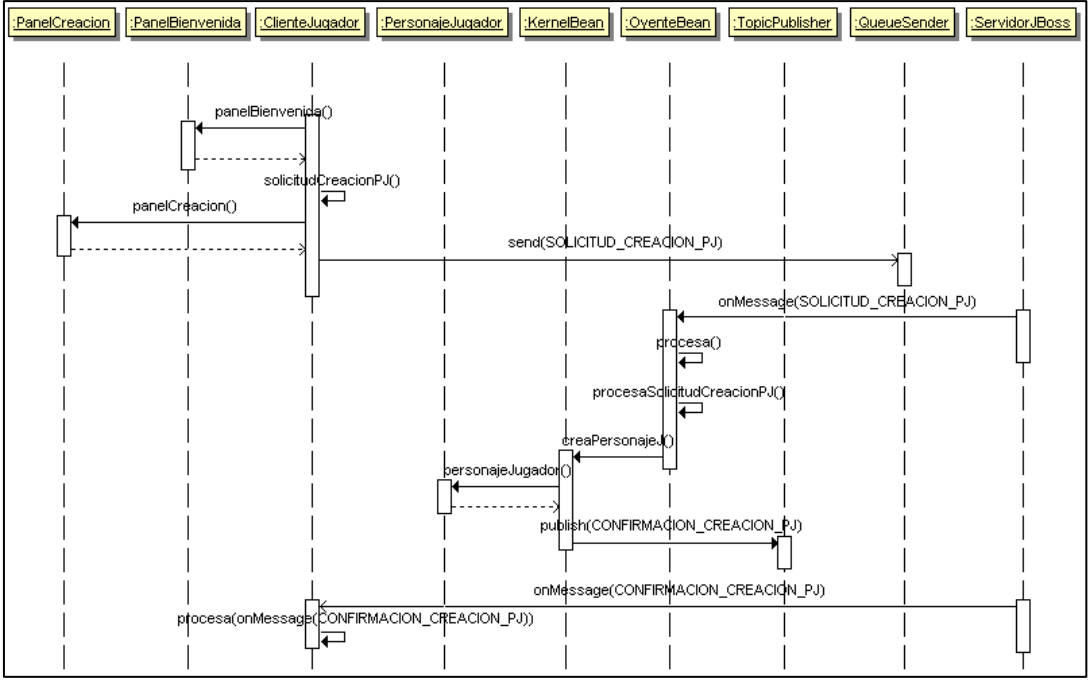
    - GestorSistemaBean ejecuta la llamada *borraTablas()* sobre **GestorSistemaDaoImpl**, el cual se encarga de acceder a la BD y borrar las tablas existentes, en caso de que ya estuvieran creadas. Para ello se apunta jdbcFactory a la BD por defecto de JBoss, se obtiene la conexión, y se va preparando y ejecutando cada query (una sentencia *drop* por cada tabla: Habitaciones, PersonajesJ, Objetos, SituacionO, Comunicacion, Inventario, PersonajesNJ).

    - GestorSistemaBean ejecuta la llamada *creaTablas()* sobre **GestorSistemaDaoImpl**, el cual se encarga de acceder a la BD y crear las tablas necesarias para el funcionamiento de la aplicación. Al igual que antes, se apunta jdbcFactory a la BD por defecto de JBoss, se obtiene la conexión, y se va preparando y ejecutando cada query (en este caso una sentencia *create*).

- Y una llamada *creaMundo()* sobre GestorSistemaBean. Aquí se crea el bean de tipo session **Kernel** que será el que proporcionará al usuario los servicios de creación y conexión del personaje, y se crean cada una de las habitaciones (Hall, Aseo, Pasillo, Conserjeria, DespachoX) con sus puertas, objetos y personajes no jugadores necesarios.

3.3.2. Crear un personaje

Diagrama de secuencia



Flujo de sucesos

- Al ejecutar **ClienteJugador** se crea la cola queue/COLA_KERNEL por la que OyenteBean recibirá las peticiones de creación, conexión y desconexión de un jugador, y se crea el panel de bienvenida. Si se pulsa el botón “Crear personaje” se llama al método *solicitudCreacionPJ*.

- Este método, haciendo uso de la clase PanelCreacion (en el paquete **InterfazGrafica**), consigue los datos necesarios para crear el personaje (id, nombre, password, imagen), los encapsula en un mensaje del tipo SOLICITUD_CREACION_PJ y lo envía a queue/COLA_KERNEL.

- **OyenteBean** recibe el mensaje a través del servidor JBoss, en su método *onMessage*. Lo envía a *procesa* y, una vez discriminado su tipo, llama a *procesaSolicitudCreacionPJ*.

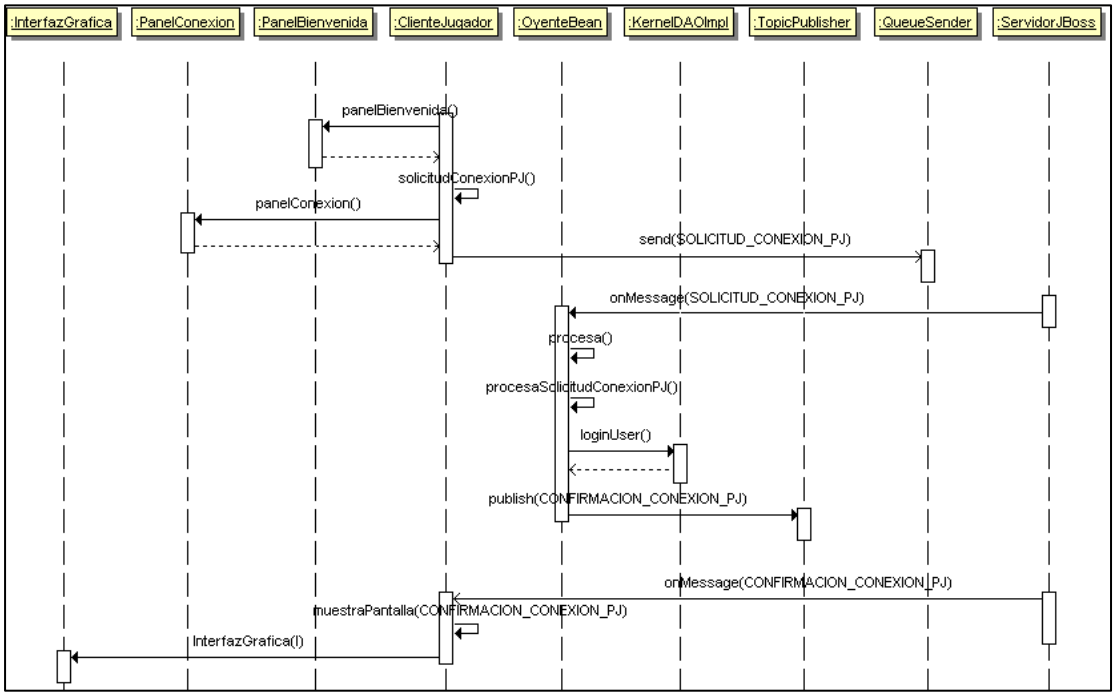
- Aquí se desencapsula el mensaje y se llama a *creaPersonajeJ* de **Kernel.** Después crea el topic CONFIRMACION (para las respuestas a las solicitudes de creación, conexión y desconexión) y envía un mensaje del tipo CONFIRMACION_CREACION_PJ.

- *creaPersonajeJ* construye un objeto de tipo **PersonajeJugador** añadiéndole sus objetos iniciales (en este caso la llave de la taquilla) y la habitación H1 (la habitación donde permanecen los personajes mientras están desconectados).

- **ClienteJugador** recibe el mensaje a través del servidor JBoss, en su método *onMessage*. Lo envía a *procesa* y muestra un mensaje de éxito o error mediante un JOptionPane.

3.3.3. Conectar un personaje

Diagrama de secuencia



Flujo de sucesos

- Al ejecutar **ClienteJugador** se crea la cola queue/COLA_KERNEL por la que OyenteBean recibirá las peticiones de creación, conexión y desconexión de un jugador, y se crea el panel de bienvenida. Si se pulsa el botón "Conectar personaje" se llama al método *solicitudConexionPJ.*

- Este método, haciendo uso de la clase PanelConexion (en el paquete **InterfazGrafica**), consigue los datos necesarios para conectar el personaje (id y password), los encapsula en un mensaje del tipo SOLICITUD_CONEXION_PJ y lo envía a queue/COLA_KERNEL.

- **OyenteBean** recibe el mensaje a través del servidor JBoss, en su método *onMessage*. Lo envía a *procesa* y, una vez discriminado su tipo, llama a *procesaSolicitudConexionPJ*.

- Aquí se desencapsula el mensaje y se llama a *loginUser* de **KernelDAOImpl** para comprobar que esos datos existen en la table personajesJ. En caso afirmativo se mete al personaje en la habitación en la que estaba cuando se desconectó y se envía al topic CONFIRMACION un mensaje del tipo CONFIRMACION_CONEXION_PJ.

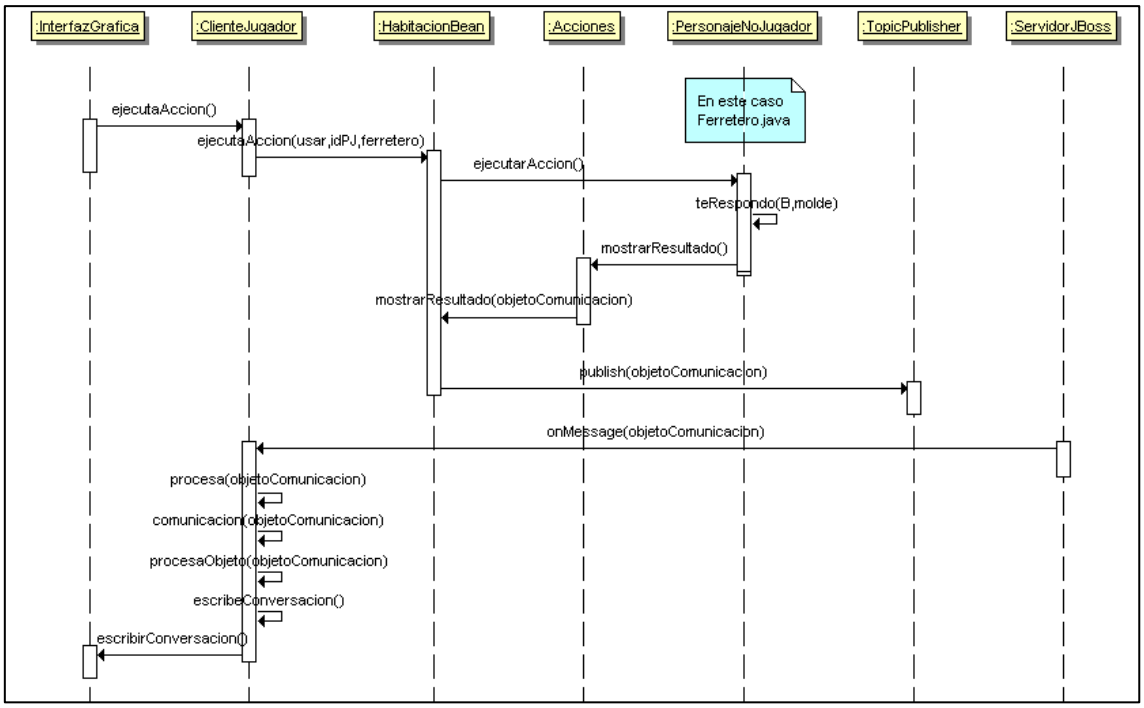
- **ClienteJugador** recibe el mensaje a través del servidor JBoss, en su método *onMessage*. Lo envía a *muestraPantalla* y se crea un objeto **InterfazGrafica**,

refrescándose la pantalla de juego y mostrándose en la consola un mensaje de éxito.

3.3.4. Ejecutar una acción (ir, coger, usar o mirar)

Utilizaremos de ejemplo de uso la acción “Usar Ferretero con jabón”.

Diagrama de secuencia



Flujo de sucesos

1. Cuando se pulsa el botón Ejecutar, se ejecuta el método *actionPerformed* de la clase OyenteEjecutar (subclase de **InterfazGrafica**). Este método actualiza los valores idAccion, objeto1IdFrase e objeto2IdFrase de **ClienteJugador** y llama a su método *ejecutaAccion*.

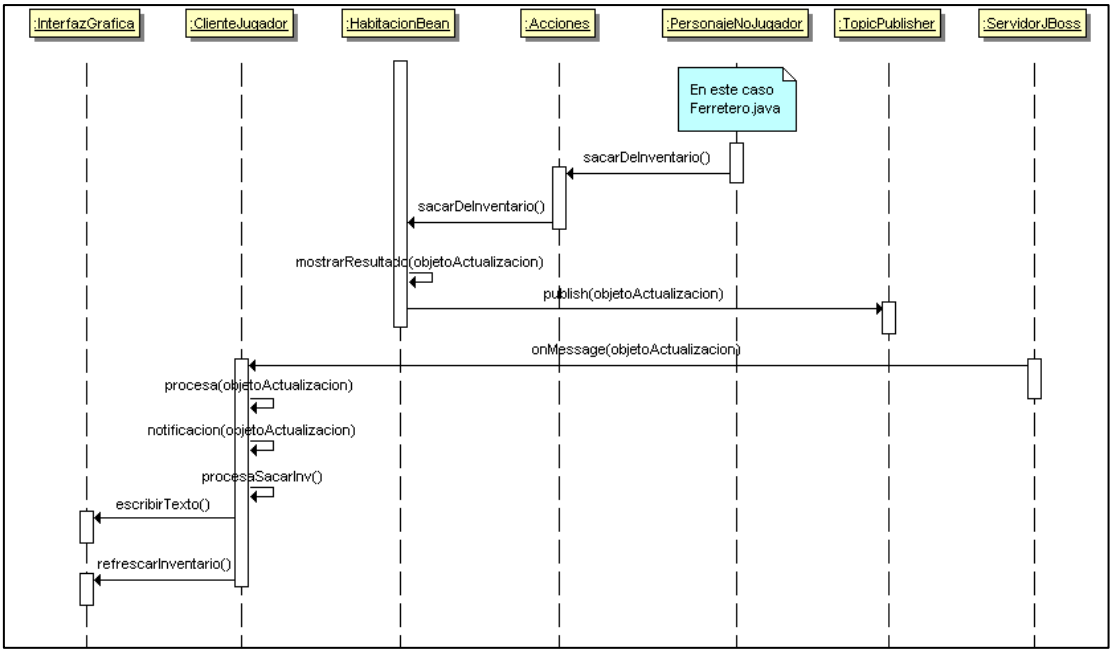
2. *ejecutaAccion* obtiene la habitación en que se encuentra el jugador y ejecuta sobre el **HabitaciónBean** (EntityBean) correspondiente la llamada *ejecutarAccion*, pasándole la acción, el jugador y el objeto.

3. Utilizando la clase Acciones, se llama al método *ejecutarAccion* de **Ferretero** (que extiende PersonajeNoJugadorClase).

4. Aquí comprueba el estado del jabón (se supone "MOLDE") y llama al método *teRespondo* (en la misma clase) con la opción "B" y el id del jugador. En este método, valiéndose del *ejecutar* de Acciones, se construye el objetoComunicación, con la conversación fija (de tipo LinkedList) y se invocan tres métodos (*mostrarResultado*, *sacarDeInventario* y *meterEnInventario*) que se explican por separado a continuación:

    - *mostrarResultado* de **Acciones**. Se le pasa como parámetro el objetoComunicacion y la habitación.

        - Llama a *mostrarResultado* de **HabitaciónBean** pasándole como parámetro el objetoComunicacion.



- Aquí, dentro del TopicSession, se crea el ObjectMessage (con el objetoComunicacion) y se le envía al topic correspondiente, en este caso **TopicPublisher**, invocando su método *publish*.

- El **ClienteJugador** de todos los personajes que se encuentran en esa habitación recibe el mensaje a través del servidor **JBoss** y lo procesa en su método *onMessage*.

- En *onMessage* se llama a *procesa*, donde se discrimina entre los distintos tipos de mensaje, hasta llamar al método *comunicación*. Aquí se desencapsula el objetoComunicacion y se le envía al método *procesaObjeto*.

- En *procesaObjeto* se discrimina si la conversación es fija o no y se llama al método *escribeConversacion*.

- En *escribeConversacion* se recurre a la clase **FragmentoConversacion** para construir la conversación (concebida como una lista de fragmentos) y una vez construida se envía al *escribirConversacion* de **InterfazGrafica**.

- Aquí se crea el objeto de la clase **EscritorTexto** (con el área de texto donde se mostrarán las conversaciones y el panel en el que estará incluida) y se lanzará un hilo (**HiloComunicacion**) para cada fragmento de la conversación. Dicho hilo llamará a *escribeTexto* de EscritorTexto que será el encargado de ir mostrando cada fragmento mediante el método *setText*.

o *sacarDeInventario* de **Acciones**. Se le pasan los siguientes parámetros: la habitación, el nombre del objeto (en este caso “JABON”), el id del jugador y el String "Me he quedado sin molde".

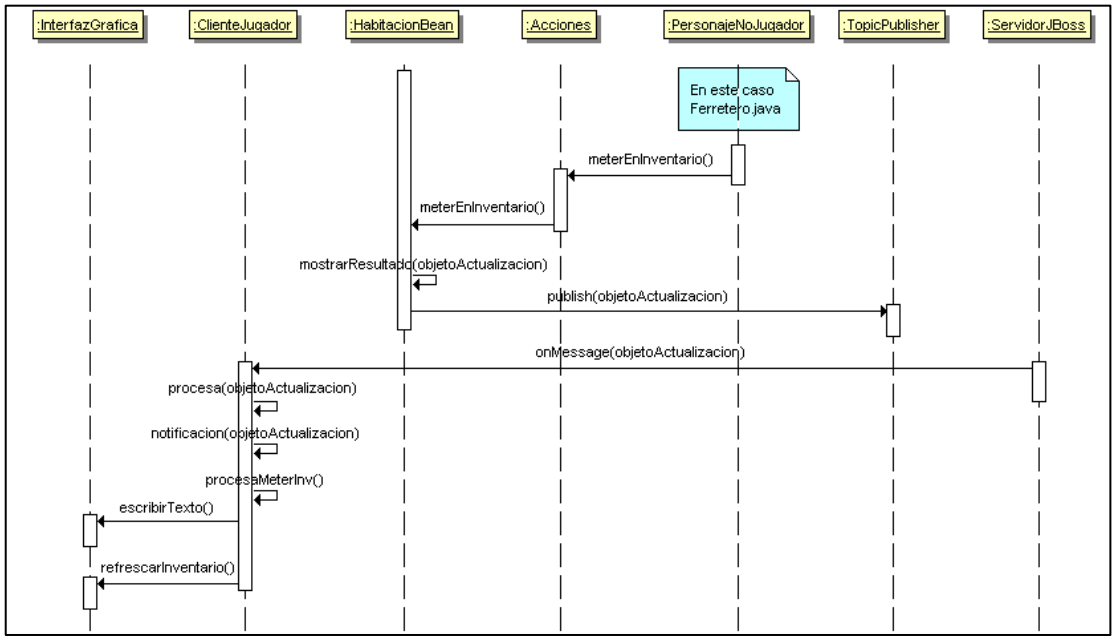
- Llama al *sacarDeInventario* de **HabitaciónBean**, donde se actualiza el atributo objetos del jugador y (valiéndose de las clases **SacarInv** y **NotificacionTextual**) construye un **ObjetoActualizacion** y se lo pasa a *mostrarResultado*.

- Aquí, dentro del TopicSession, se crea el ObjectMessage (con el objetoActualizacion) y se le envía al topic correspondiente, en este caso **TopicPublisher**, invocando su método *publish*.

- El **ClienteJugador** de todos los personajes que se encuentran en esa habitación recibe el mensaje a través del servidor **JBoss** y lo procesa en su método *onMessage*.

- En *onMessage* se llama a *procesa*, donde se discrimina entre los distintos tipos de mensaje, hasta llamar al método *notificación*. Aquí se desencapsula el objetoActualizacion y se le envía (convertido a la clase **SacarInv**) al método *procesaSacarInv*.

- Aquí se llama a dos métodos de InterfazGrafica: *escribirTexto* (con el String "Me he quedado sin molde") y a *refrescarInventario* (que borra los objetos que tuviera el jugador y los vuelve a pintar).



- *meterEnInventario* de **Acciones**. Se le pasan los siguientes parámetros: la habitación, el tipo del objeto (en este caso llave), el id del jugador y el String "He conseguido una llave".

- Llama al *meterEnInventario* de **HabitaciónBean**, donde se actualiza el atributo objetos del jugador y (valiéndose de las clases **MeterInv** y **NotificacionTextual**) construye un **ObjetoActualizacion** y se lo pasa a *mostrarResultado*.

- Aquí, dentro del TopicSession, se crea el ObjectMessage (con el objetoActualizacion) y se le envía al topic correspondiente, en este caso **TopicPublisher**, invocando su método *publish*.

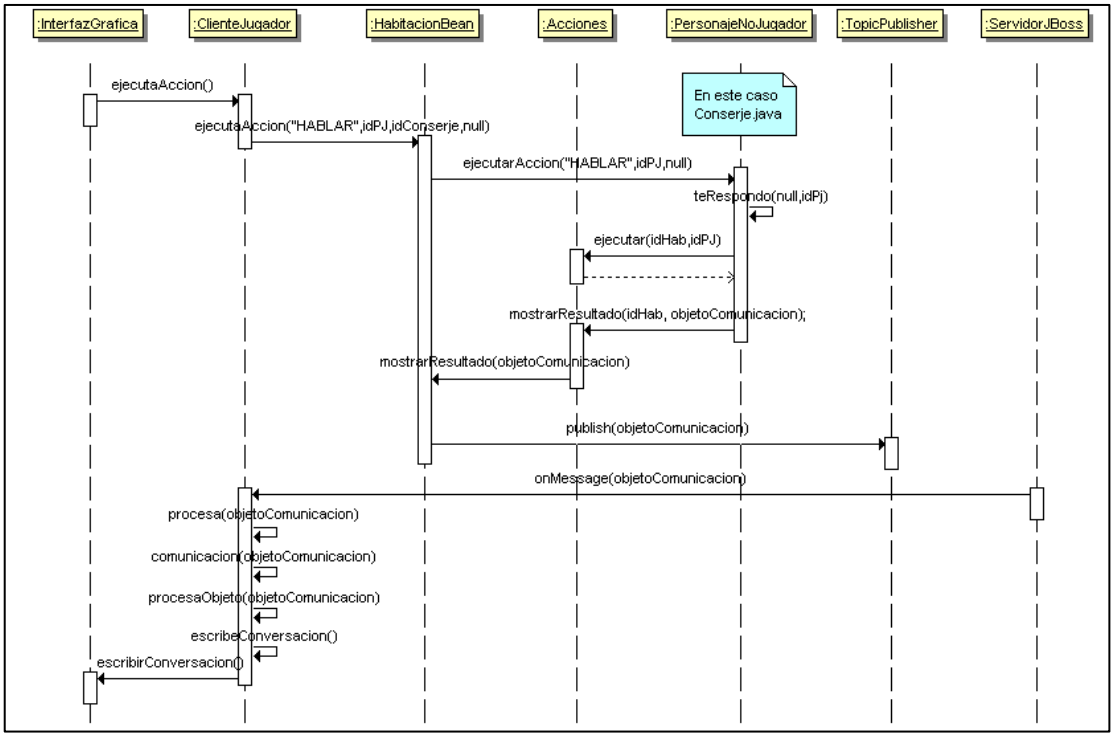
- El **ClienteJugador** de todos los personajes que se encuentran en esa habitación recibe el mensaje a través del servidor **JBoss** y lo procesa en su método *onMessage*.

- En *onMessage* se llama a *procesa*, donde se discrimina entre los distintos tipos de mensaje, hasta llamar al método *notificación*. Aquí se desencapsula el objetoActualizacion y se le envía (convertido a la clase **MeterInv**) al método *procesaMeterInv*.

- Aquí se llama a dos métodos de InterfazGrafica: *escribirTexto* (con el String "He conseguido una llave") y a *refrescarInventario* (que borra los objetos que tuviera el jugador y los vuelve a pintar).

3.3.5. Hablar con un personaje

Diagrama de secuencia



Flujo de sucesos

1. Cuando se pulsa el botón Ejecutar, se ejecuta el método *actionPerformed* de la clase OyenteEjecutar (subclase de **InterfazGrafica**). Este método actualiza los valores idAccion, objeto1IdFrase e objeto2IdFrase de **ClienteJugador** y llama a su método *ejecutaAccion*.

2. *ejecutaAccion* obtiene la habitación en que se encuentra el jugador y ejecuta sobre el **HabitaciónBean** correspondiente la llamada *ejecutarAccion*, pasándole la acción, el jugador y el objeto.

3. Utilizando la clase Acciones, se llama al método *ejecutarAccion* de **Conserje** (que extiende PersonajeNoJugadorClase).

4. Este método llama al método *teRespondo* (en la misma clase). En este método, valiéndose del *ejecutar* de Acciones, se construye el objetoComunicación, con la conversación fija y con las opciones de respuesta (ambas conversaciones de tipo LinkedList). Una vez construido dicho objeto se invoca al método *mostrarResultado* de **Acciones**, pasándole como parámetro el objetoComunicacion y la habitación.

5. *mostrarResultado* de Acciones llama a *mostrarResultado* del **HabitacionBean** correspondiente.

6. Aquí, dentro del TopicSession, se crea el ObjectMessage (con el objetoComunicacion) y se le envía al topic correspondiente, en este caso **TopicPublisher**, invocando su método *publish*.

7. El **ClienteJugador** de todos los personajes que se encuentran en esa habitación recibe el mensaje a través del servidor **JBoss** y lo procesa en su método *onMessage*.

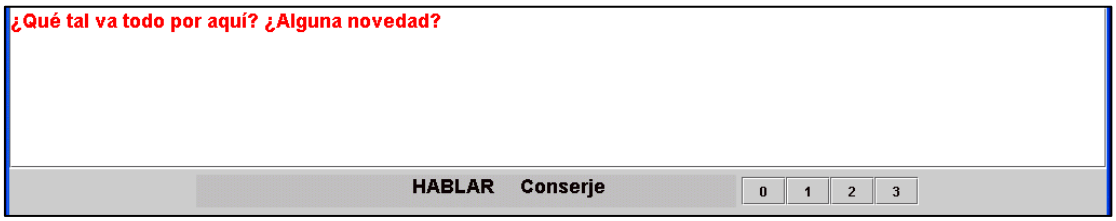
8. En *onMessage* se llama a *procesa*, donde se discrimina entre los distintos tipos de mensaje, hasta llamar al método *comunicación*. Aquí se desencapsula el objetoComunicacion y se le envía al método *procesaObjeto*.

9. En *procesaObjeto* se discrimina si la conversación ha finalizado o no. En este caso se está comenzado, por tanto se deshabilitan los botones ejecutar, envioChat y desconectar de la interfaz grafica (para que no se pueda interrumpir una conversación) y se llama al método *escribeConversacion*.

10. En *escribeConversacion* se recurre a la clase **FragmentoConversacion** para construir la conversación (concebida como una lista de fragmentos) y una vez construida se envía al *escribirConversacion* de **InterfazGrafica**.

11. Aquí se crea el objeto de la clase **EscritorTexto** (con el área de texto donde se mostrarán las conversaciones y el panel en el que estará incluida) y se lanzará un hilo (**HiloComunicacion**) para cada fragmento de la conversación. Dicho hilo llamará a *escribeTexto* de EscritorTexto que será el encargado de ir mostrando cada fragmento mediante el método *setText* y de mostrar u ocultar (si la conversación global ha terminado) la botonera de opciones de respuesta al llegar a la última línea de la conversación.

12. Una vez que se muestra la conversación fija se muestra la conversación de respuesta, apareciendo pues una botonera con los distintos botones de respuesta, cada uno de ellos enlazado a su correspondiente opción:



3.163. Screanshot de la consola y el display con la botonera de opciones

El flujo de sucesos que tienen lugar al responder es el siguiente:

13. Las diversas opciones de respuesta van apareciendo y desapareciendo en la consola al situarse el ratón sobre cada una de ellas o salirse, esto es así por los métodos *mouseEntered* y *mouseExited* de la clase OyenteOpcionMouse (subclase de **InterfazGrafica**). Cuando se pulsa cualquiera de ellos se ejecuta *mouseClicked* que llama al método *teRespondo* de **ClienteJugador**, pasándole como parámetro la opción de respuesta elegida (opcionInterna).

14. *teRespondo* obtiene la habitación en que se encuentra el jugador y ejecuta sobre el **HabitaciónBean** correspondiente la llamada *teRespondo*, pasándole la opción, el jugador y el personaje no jugador con él que se está hablando.

15. Utilizando la clase Acciones, se llama al método *teRespondo* de **Conserje** (que extiende PersonajeNoJugadorClase) pasándole la opción y el jugador.

16. En este método se cambia el estado del conserje de "ESPERANDO" a "CONVERSANDO", se discrimina en base a la opción de respuesta elegida y, del mismo modo que antes, se construye el objetoComunicación y se invoca al método *mostrarResultado* de **Acciones**, pasándoselo como parámetro junto a la habitación.

17. A partir de aquí, el flujo de sucesos que tienen lugar hasta que se crea el mensaje y llega de nuevo a *procesa* de **ClienteJugador** son los mismos que vimos al principio (ver puntos 5-11).

Veremos que pasa cuando la conversación termina:

- Cuando en la botonera de opciones de la **InterfazGrafica** hemos elegido una opción que va a desencadenar el fin de la conversación, los pasos hasta que se crea el objeto son los mismos (ver puntos 13-15).

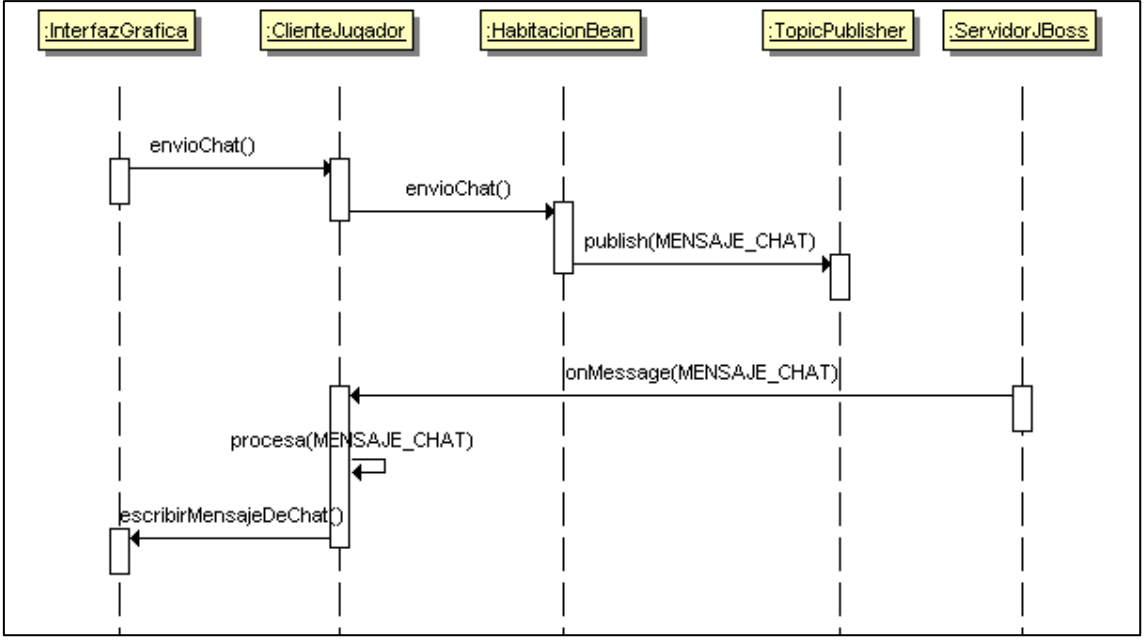
- La diferencia viene al llegar a *teRespondo* de **Conserje** donde se discrimina en base a la opción de respuesta elegida y se construye el objetoComunicación, que en este caso sólo incluirá un LinkedList con la conversación fija que se va a desencadenar.

- Los pasos que se siguen una vez creado el mensaje hasta que llega a *procesa* de **ClienteJugador** son los mismos que vimos al principio (ver puntos 5-8).

- Es en *procesaObjeto* donde se discrimina si la conversación ha finalizado o no. Al no haber un LinkedList con las opciones de respuesta se oculta la botonera de opciones y se llama al método *escribeConversacion*, que en este caso sólo escribirá la conversación fija, desapareciendo la consola al llegar a la última línea, y quedando la pantalla de juego tal y como estaba antes de empezar la conversación.

3.3.6. Hablar con otros jugadores

Diagrama de secuencia



Flujo de sucesos

- Cuando se pulsa el botón Enviar, se ejecuta el método *actionPerformed* de la clase OyenteEnvioMsgChat (subclase de **InterfazGrafica**). Este método borra el texto escrito en el JTextArea saliente y se lo pasa como parámetro al método *envioChat* de **ClienteJugador**.

- *envioChat* obtiene la habitación en que se encuentra el jugador y ejecuta sobre el **HabitaciónBean** correspondiente la llamada *envioChat*, pasándole el mensaje y el nombre del jugador.

- Aquí, dentro del TopicSession, se crea el Message (de tipo "MENSAJE_CHAT") y se le envía al topic correspondiente, en este caso **TopicPublisher**, invocando su método *publish*.

- El **ClienteJugador** de todos los personajes que se encuentran en esa habitación recibe el mensaje a través del servidor **JBoss** y lo procesa en su método *onMessage*.

- En *onMessage* se llama a *procesa*, donde se discrimina entre los distintos tipos de mensaje, hasta llamar al método *escribirMensajeDeChat* de **InterfazGrafica** (al que se le pasan como parámetros el texto y el nombre del jugador desencapsulados).

- Este método será el encargado de volcar en el JTextArea mensajesEntrantes el texto precedido del nombre del jugador que lo escribió utilizando el método *append*.

3.4. Integración de animación

Dados los problemas para revivir la aplicación, los surgidos durante la reingeniería de la interfaz y el tiempo que conllevaron el entendimiento global de la aplicación y su documentación, los avances en la integración de la animación han sido menores de lo deseado.

Esta parte se dejó para el final porque para integrar la animación era necesario primero entender el funcionamiento de la aplicación y segundo que la interfaz gráfica nueva fuera completamente estable. Una vez conseguidos ambos objetivos y tras integrar en la nueva interfaz la solución a algunos bugs (como el de que se perdieran mensajes en las conversaciones) se estuvo en condiciones de abordar la integración de la animación.

El primer paso fue integrar la nueva interfaz a los paquetes animación, animación.interfazGrafica y animación.interfazGrafica.imagenes, los cuales fueron renombrados como agm.clienteJugador, agm.clienteJugador.interfazGrafica y agm.clienteJugador.interfazGrafica.imagenes.

En este punto, se procede a arrancar la aplicación pero al conectar el jugador se produce el siguiente error:

```
******ClienteJugador:PRocesando CONFIRMACION_CONEXION_PJ
entrando en interpretaMascara
0.400-740.520
posXRelativa = 630
posYRelativa = 27
PERSONAJEJUGADOR:  pJ.getPosX() = 630; pJ.getPosY() = 427
Uncaught error fetching image:
java.lang.NullPointerException
        at sun.awt.image.URLImageSource.getConnection(URLImageSource.java:99)
        at sun.awt.image.URLImageSource.getDecoder(URLImageSource.java:108)
        at sun.awt.image.InputStreamImageSource.doFetch(InputStreamImageSource.java:248)
        at sun.awt.image.ImageFetcher.fetchloop(ImageFetcher.java:172)
        at sun.awt.image.ImageFetcher.run(ImageFetcher.java:136)
```

Relacionado con la integración de la animación, en la documentación del año 2004/05 se podía leer lo siguiente:

4.5. Integración con AGM

Por falta de tiempo, y problemas surgidos en la detección de colisión contra los bordes, cuya solución llevó más tiempo del deseable, la integración de la animación en la aplicación final no ha podido realizarse más que parcialmente (se ha incluido un nuevo paquete con las clases ya modificadas). En el momento actual en que se redactó este documento, se estaba realizando un proceso de depuración de las clases integradas, para así dotar a la aplicación de una animación ya integrada en sus aspectos más básicos.

Sin embargo, se proporcionará en el presente apartado una serie de recomendaciones a seguir en el momento de continuar dicha integración:

Pantalla de la aplicación:
Las funciones de la pantalla principal (la ventana del juego) en la AGM las desempeña la clase Bot. En este caso, bastaría en principio con modificar la clase adecuadamente, añadiendo la mayoría de los métodos de la ventana principal del programa de pruebas.

Pendiente de implementación:
ActualizarPantalla() requiere de algunas modificaciones (vienen indicadas como texto comentado).

ObjetoEnPantalla:
Esta clase será la candidata a extender las capacidades de un sprite. En principio, a diferencia de lo que ocurría en el programa de pruebas, esta clase no será abstracta, y todos los objetos que se representen serán instancias de ella, es decir, tendrán todas la misma estructura. Si resultara más recomendable que, por ejemplo, objetos y personajes dispusieran de una estructura más heterogénea, podría definirse una jerarquía de clases que extendieran a ésta.

Pendiente de implementación:
Restan por añadir métodos de cambio de estado, correcciones en el método moverse(X,Y) y tal vez algunas ampliaciones.

InterfazGrafica:
Adaptar esta clase a la nueva clase ObjetoEnPantalla no parece tampoco en principio una tarea complicada. Principalmente, consistiría en revisar las apariciones de instancias de ObjetoEnPantalla y adecuarlas a los posibles cambios.

ClienteJugador:
Modificar esta clase consistiría también en una revisión de lo ya existente. Debe tenerse cuidado con los cambios de estado y la respuesta ante distintas acciones.

Pendiente de Implementación:
Además, si queremos, por ejemplo, que al ejecutar una acción "Coger un objeto X" se visualice al personaje desplazándose hasta la posición del objeto y que no se ejecute la acción en sí de coger el objeto hasta que el personaje ha llegado, deberíamos incluir mecanismos de comunicación entre los objetos en pantalla existentes y la aplicación, de forma que cuando el sprite se encuentre suficientemente cercano al objeto envíe una señal a la aplicación (que tendría que ponerse de algún modo "a la escucha", y permitir dedicarse mientras tanto a otra cosa, como el intercambio de mensajes con otros jugadores, o directamente "dormirse") para que se ejecute la acción de forma efectiva. Aquí es donde pueden presentarse problemas.

Serán precisos también mecanismos de comunicación entre instancias de la aplicación, para permitir la correcta actualización de la interfaz de usuario de todos los usuarios. Es decir, que un PJ efectúe un movimiento de X a Y, y esta acción se vea reflejada en las pantallas de todos los demás jugadores

Asimismo, debería comprobarse la utilidad de la máscara.

Resto de clases

Por el momento, no se ve que existan problemas con las demás clases del proyecto. En caso contrario, habría que estudiarlos con un mayor detenimiento.

Lo que se ha conseguido en este punto es integrar en la nueva interfaz esos paquetes con las clases modificadas a las que se hace alusión en el texto anterior con lo que siguen pendientes de implementación los mismos métodos que quedaban entonces pero ahora sobre una interfaz más fácil de entender.

3.5. Trabajo futuro

- Queda pendiente el problema de las distintas configuraciones ya que actualmente, y al contrario de lo que pudiera desprenderse de las memorias de años anteriores, sólo funciona para una resolución de pantalla de 1024x768 px.

Como ya se ha comentado, las clases **ConfigIG_1024_768**, **ConfigIG_800_600** y **ConfigIG** han sido suprimidas pues lo único que hacían eran fijar unos parámetros para cada configuración, que no se utilizaban (siempre se pasaban los mismos: *iG = new InterfazGrafica(this, InterfazGrafica.r1024_768);*) y que de utilizarse serían insuficientes, cuando no incorrectos.

Para implementar correctamente el cambio de configuración habría que calcular un factor para cada eje de referencia el cual sería pasado como parámetro a cada método encargado de crear y dimensionar los distintos componentes de la interfaz (pantalla de juego, consola, display, botonera de acciones, inventario y chat).

Además habría que adaptar también el tamaño de los objetos en pantalla (objetos no personajes, personajes no jugadores y personajes jugadores) así como las coordenadas en las que aparecen.

Hilando muy fino se deberían cambiar también el tamaño de las fuentes y las imágenes del cambio de puntero.

Todo esto se puede hacer pero tal vez la solución óptima sería cambiar la resolución del usuario mediante código utilizando la librería User32.dll y ejecutar en modo full-screen (pantalla completa) como se hacía en la versión MS-DOS.

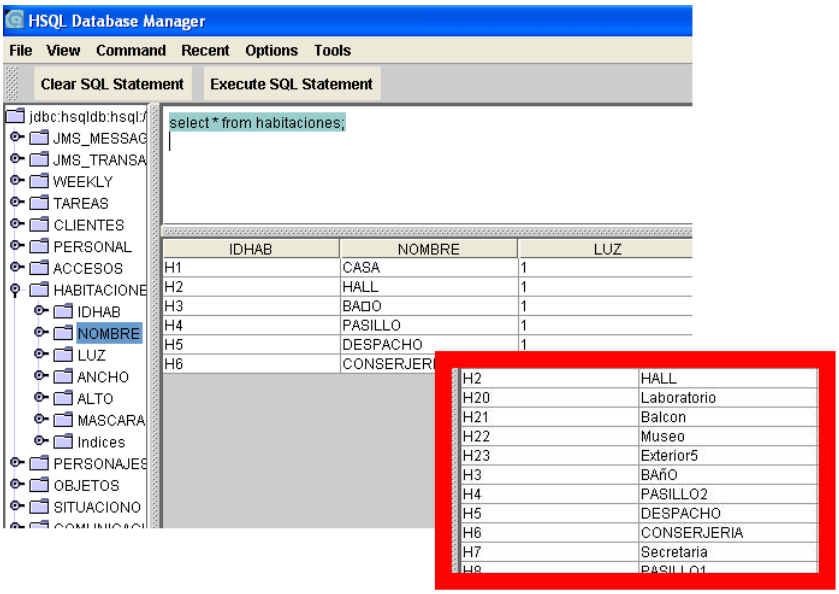
- Problema con caracteres especiales.

Algunos problemas relacionados con los caracteres especiales fueron solucionados cambiando el texto a mostrar a nivel de código (así, en algunas líneas aparecían palabras como aplicaciÃ³n en lugar de aplicación).

Lo que no se pudo arreglar fue que se mostraran correctamente aquellas palabras que provenían de la base de datos. Por ejemplo, aún reescribiendo el nombre que se da a la habitación BAÑO en crearAseo() (GestorSistemaBean.java) seguía apareciendo de la siguiente forma:

3.24. Problema con los caracteres especiales

Tras comprobar mediante el DataBaseManager que el nombre de la habitación se guardaba de manera defectuosa en la base de datos (ver figura de abajo) y al tratarse de un caso particular que no afectaba al resto del equipo se decidió pasar por alto este problema que seguramente esté relacionado con la codificación que usa la versión del jdk o la GUI del HSQLDB (se aconseja estandarizar ISO-Latin 8859).



3.25. Comparación entre dos gestores de la base de datos

- Relacionada con el display, una mejora podría ser la búsqueda de una solución óptima para las comunicaciones entre jugador y personajes ya que la actual, que elige las posibles respuestas mediante una fila de botones y la muestra en el componente inferior de un **JOptionPane** parece cuanto menos mejorable (se intentó poner un fondo transparente a la consola infructuosamente).

  Podría implementarse la solución elegida en la versión MS-DOS consistente en mostrar unos bocadillos (para que quede más claro quien está hablando), que cambiara de color la respuesta según se pasara el ratón por encima de cada una de ellas y que se seleccionaran haciendo click.

- Otra mejora relacionada con el display podría ser sustituir la funcionalidad del botón Ejecutar por un evento click.

  Esta mejora fue iniciada con el cambio de puntero al elegir una acción pero no pudo concluirse por falta de tiempo. En la versión original para “mirar el conserje” había que hacer clic en “Mirar”, luego hacer clic en “Conserje” y luego hacer clic en “Ejecutar”. Tal y como está ahora se pulsa “Mirar” (el puntero cambia de forma y se pone “Mirar” en el display), se hace clic en “Conserje” (se pone “Mirar conserje” en el display) y se hace click en Ejecutar. El objetivo sería que bastase con hacer clic en “Mirar” y, con el puntero con forma de lupa, hacer clic sobre el conserje.

- Relacionado con la botonera de acciones, actualmente el puntero adopta la forma de la acción escogida cuando entra en la pantalla de juego pero, al igual que ocurre en la versión MS-DOS, no sería demasiado difícil implementar esto de tal

modo que sólo se produjera el cambio al situarse sobre un objeto o personaje sobre el que tuviera sentido dicha acción.

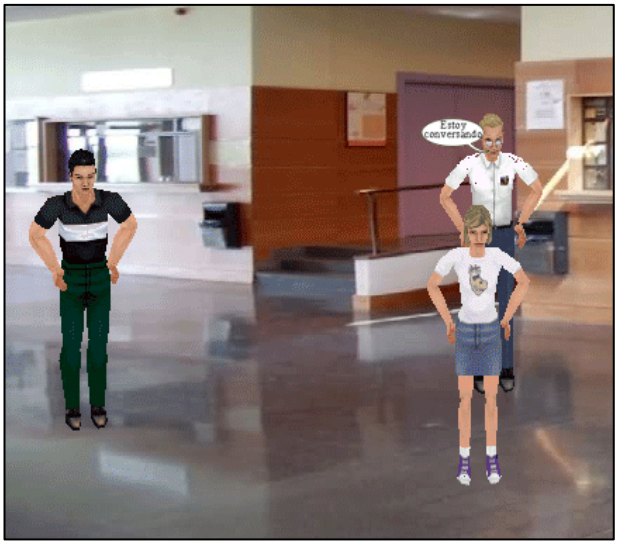
Idealmente estaría bien capar acciones de todo punto ilógicas como ocurría en la versión MS-DOS (ejemplo: cerrar expediente).

- Al igual que ocurre con la botonera de acciones, una posible mejora en torno al inventario podría ser implementar el cambio de puntero al hacer click sobre un determinado objeto (como ocurría en la versión MS-DOS). Si se quisiera evitar el tener que clickear bastaría con sustituir los objetos del inventario por componentes JLabel y añadirles un MouseListener.

  Otra mejora podría ser añadirle un controlador GridLayout(2,4) para que no haga falta utilizar las scrollbars hasta que no se tengan ocho objetos.

- Del mismo modo que se cambió la imagen de la llave de la taquilla y posteriormente la llave del despacho para que no fueran iguales, al cambiar la imagen del conserje conversando (para que no fuera igual a la del conserje esperando) se detectó un bug, imperceptible hasta ese momento, relacionado con el cambio de imágenes asociado a un cambio de estado.

En este caso, cuando un jugador está conversando con el conserje el otro no se da cuenta hasta que no sale del hall y vuelve a entrar. Además, para el jugador que está conversando con él la imagen no cambia.



3.26. Conserje conversando con un jugador

Si un jugador cambia el estado de una habitación y dicho cambio ha de ser visible al resto de jugadores que se encuentren en ella (como es el caso), la secuencia de acciones que desencadena el actor han de concluir en un refresco para todas esas pantallas. Por falta de tiempo no ha podido solucionarse este error pero no parece demasiado complicado.

4. Framework para la creación de historias en AGM

4.1. Resumen introductorio

La idea de desarrollar un editor de historias para AGM surgió ante la necesidad de saber programar en java y conocer la aplicación para poder realizar una misión.
En el editor por lo tanto supone un modo sencillo y rápido para crear misiones además de una herramienta para control y modificación de las mismas.

El editor consta de una interfaz gráfica desarrollada en Java bajo Eclipse en la que podemos crear una nueva misión, cargar una misión ya existente, guardar una misión e importar una misión a AGM. En esta misma interfaz también se muestran los objetos, personajes y habitaciones y todas las acciones que se pueden realizar con los mismos.
Lo que se conseguirá tras introducir datos en esta interfaz será un fichero de extensión .agm en el que se guardará la misión creada, y una fichero de texto de nombre misiones.txt en el directorio principal del proyecto en cuya primera línea aparecerá la ruta en la que se encuentra dicho fichero .agm para que pueda ser cargado posteriormente desde AGM.

Para la conexión del Editor con AGM se han utilizado dos clases principales: ObjetoEditor y PersonajeEditor. Estas dos clases son una defnición genérica de todos los objetos y personajes editados respectivamente, completada con los datos obtenidos mediante el editor.
Además, se han añadido métodos para crear las habitaciones, puertas, objetos y personajes editados dentro de los métodos utilizados en la aplicación original AGM para crear el mundo.

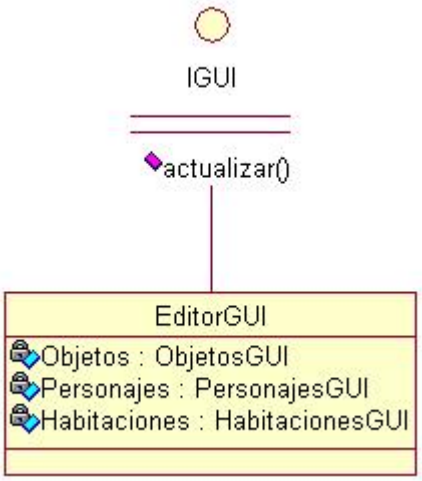
4.2. Diagramas de Clases

A continuación exponemos los diagramas de clases separados por paquetes puesto que no cabe la imagen de todos juntos. Iremos explicando el sistema en sentido top-down comenzando por las clases fundamentales del paquete vista y mostrando poco a poco las clases que las implementan, posteriormente se explicará el controlador y la fachada y la clase Transfer para que la explicación del paquete modelo sea más entendible, y por último finalizaremos con el paquete integración.

4.2.1. Paquete vista

En primer lugar contamos con la inteface IGUI la cual contiene el método actualizar() que será implementado por las interfaces principales: EditorGUI, ObjetosGUI, PersonajeGUI y HabitacionesGUI. Este método servirá para poder actualizar las interfaces en las que se realizó una acción y lo explicaremos más en profundidad en el aparatado 4.2.2 referente al Controlador.

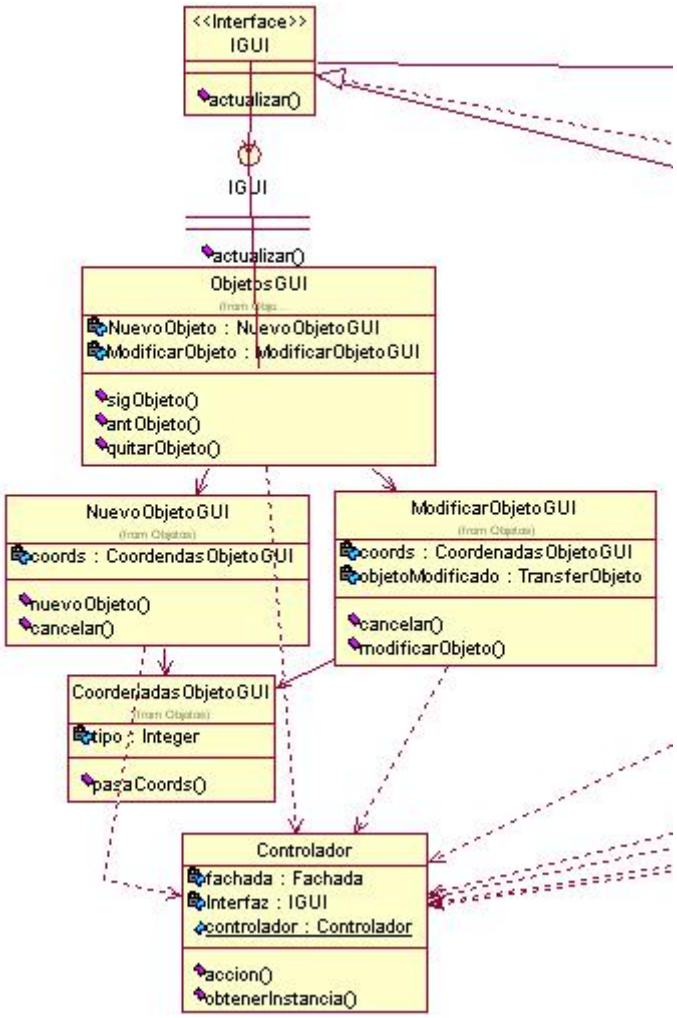
También se encuentra en el paquete vista la clase EditorGUI que será la interfaz principal del editor de misiones y que como indicabamos antes implementa la interfaz IGUI. EditorGUI contiene botones para acceder a la interfaz de objetos, a la de personajes y a la de habitaciones, también dispone de botones que nos permitirán crear una nueva misión, cargar una misión, guardar una misión e importar una misión.
Se puede ver el diagrama de clases de la interface IGUI y de la clase EditorGUI en la imagen 4.01.
A continuación pasamos a explicar las distintas interfaces a las que podemos acceder desde EditorGUI.



4.01 Diagrama general del paquete vista

4.2.1.1. Paquete vista.objetos

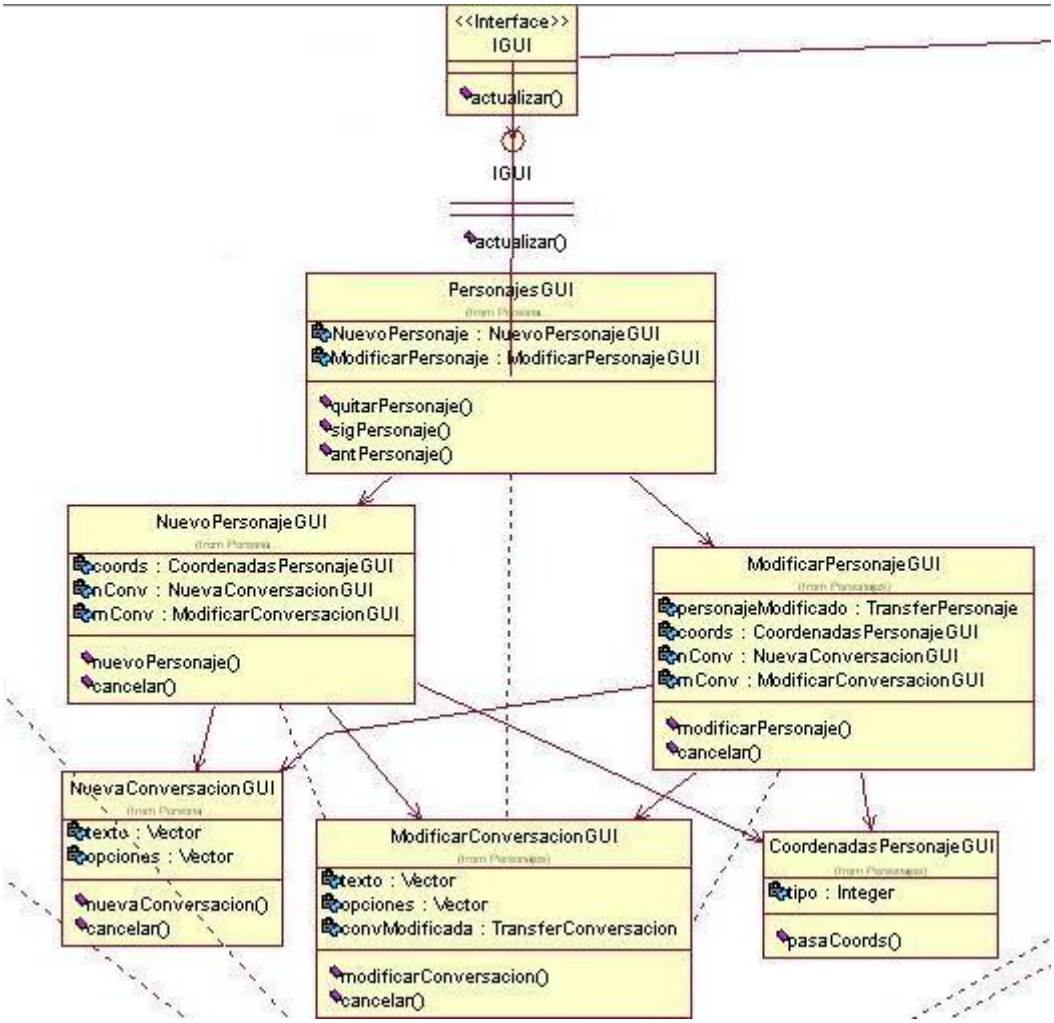
La clase ObjetosGUI implementan la interface IGUI y contiene a su vez dos interfaces graficas más a las que se podrá acceder: NuevoObjetoGUI y ModificarObjetoGUI desde las que se obtendrán los datos relativos a un objeto.
Tanto NuevoObjetoGUI como ModificarObjetoGUI contienen un objeto del tipo CoordenadasObjetoGUI que es otra interfaz gráfica en la que podremos capturar las coordenadas de la posición en la que se haya clickeado el ratón.
El diagrama de estas cuatro clases se puede ver en la imagen 4.02.
Los métodos sigObjeto() y antObjeto() de ObjetoGUI nos servirán para mostrar en la interfaz de los objetos el objeto anterior al que se está mostrando y el posterior respectivamente, ya que tanto quitarObjeto() como el TransferObjeto que se pasa a ModificarObjetoGUI hacen referencia al objeto que se está mostrando en la interfaz principal de objetos en el momento en que se pulsa el botón correspondiente. La variable tipo de CoordenadasObjetoGUI servirá para diferenciar si venimos de NuevoObjetoGUI o desde ModificarObjetoGUI para posteriormente poder actualizar bien la interfaz correspondiente.



4.02 Diagrama de clases del paquete vista.objetos

4.2.1.2. Paquete vista.personajes

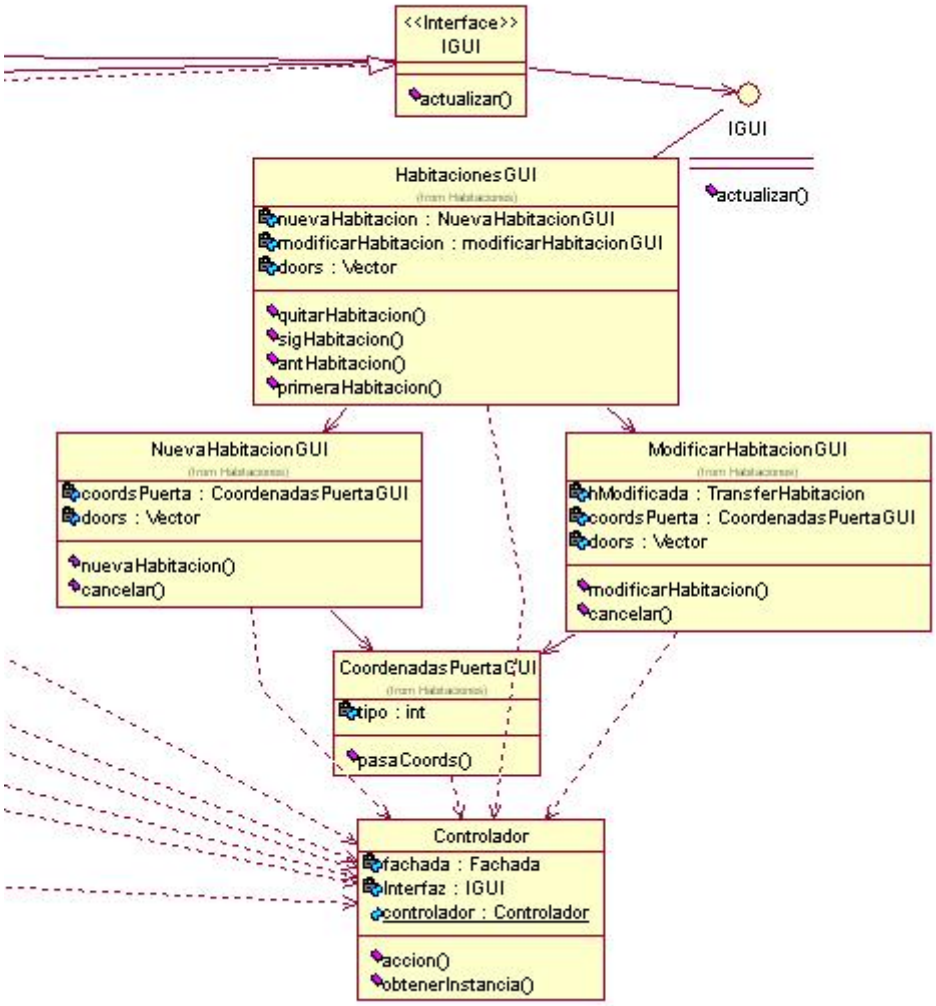
Como se puede apreciar en el diagrama de la imagen 4.03, el paquete Personajes está compuesto de manera similar al de los objetos con la salvedad de que contiene las GUIs necesarias para crear conversaciones, ya que éstas estarán asociadas a los personajes desde los cuales se crean. Se pueden crear y modificar conversaciones tanto desde NuevoPersonajeGUI como desde ModificarGUI ya que ambas clases contienen los objetos nConv y mConv del tipo NuevaConversacionGUI y ModificarConversacionGUI respectivamente, que son interfaces gráficas desde las cuales obtenemos los datos necesarios para crear una conversación.
La variable tipo de CoordenadasPersonajeGUI tiene la misma función que en el caso de los objetos.



4.03 Diagrama de clases del paquete vista.personajes

4.2.1.3. Paquete vista.habitaciones

El paquete vista.habitaciones tiene una estructura idéntica al de los objetos, como se puede apreciar en el diagrama de la imagen 4.04, puesto que las acciones que se realizan en ambos son similares. La única diferencia respecto a los objetos es, aparte por supuesto de los datos necesarios para crear los elementos en cada caso, el uso del Vector doors tanto en NuevaHabitacionGUI como en ModificarHabitacionGUI en el cual iremos almacenando las puertas que se creen en una habitación.
Nuevamente contamos con la variable tipo en CoordenadasPuertaGUI para diferenciar la interfaz de la que venimos.



4.04 Diagrama de clases del paquete vista.habitaciones

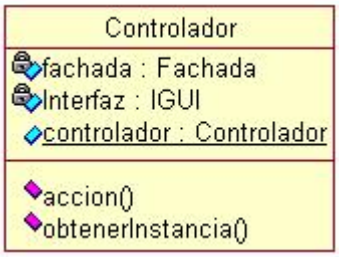
Aparte de los tres módulos principales del paquete vista hay que indicar que las clases para filtros de imágenes y misiones así como la clase de preview de imágenes y la clase donde están definidos todos los eventos se encuentran en el paquete principal vista.

4.2.2. Paquete controlador

Como comentabamos en el apartado 4.2.1 referente al paquete vista, la interface IGUI contiene el método actualizar() que sirve para poder actualizar la interfaz desde la cual se realizó alguna acción. Esta actualización de la interfaz se inicia desde el controlador tras haber recibido un evento desde las interfaces gráficas y haber realizado la acción correspodiente a dicho evento.

El Controlador por lo tanto, recibirá eventos, delegará en la Fachada para realizar las acciones convenientes y mandará un evento a la interfaz principal para que se encargue a su vez de mandar el evento a la interfaz original en la que se inició la acción.

Por lo tanto, como podemos ver en la imagen 4.05 el Controlador estará compuesto de una Fachada para poder realizar las acciones, de un atributo de tipo IGUI que será la interfaz principal del editor para poder actualizar las interfaces y de un atributo del propio tipo Controlador puesto que se accederá a él de forma estática mediante el método obtenerInstancia(). También contendrá un método accion() que se utilizará desde las interfaces gráficas para lanzar un evento al Controlador cuando se realice alguna acción.



4.05 Diagrama de clase del Controlador

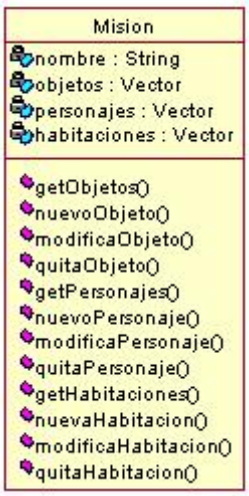
Una vez tenemos definido el paquete vista y el Controlador pasamos a explicar el paquete modelo.

4.2.3. Paquete modelo

El paquete modelo contiene la fachada compuesta por la interface Fachada y la clase FachadaImp, las factorías para crear los objetos, la interface Transfer y subpaquetes para cada módulo de la aplicación (objetos, personajes y habitaciones) que contienen el Transfer relativo al subpaquete y la clase que gestiona los datos del elemento.

Los Transfer contienen todos los datos relativos a un elemento, son TransferObjeto, TransferPersonaje, TransferHabitación, TransferPuerta y TransferConversación las cinco clases implementan la interface Transfer que no contiene nada.
Profundizaremos más en cada uno de los Transfer y los atributos que lo componen en los apartados correspondientes a cada transfer: *4.2.3.4 Transfer Objeto, 4.2.3.5 Transfer Personaje, 4.2.3.6 Transfer Conversación, 4.2.3.7 Transfer Habitación* y *4.2.3.8 Transfer Puerta*.

En el paquete modelo también tenemos la clase Misión (imagen 4.06). Esta clase será la que escribamos como objeto en el fichero .agm de la misión, por lo tanto implementa Serializable para poder ser escrita como objeto directamente sobre un fichero.
La clase misión está formada por el nombre de la misión y tres vectores que contienen los objetos, personajes y habitaciones que conforman la misión.
Además la clase misión contiene métodos para poder añadir, modificar o eliminar directamente valores de cualquiera de los vectores para posteriormente facilitar el proceso de acceso a los datos en los Daos.



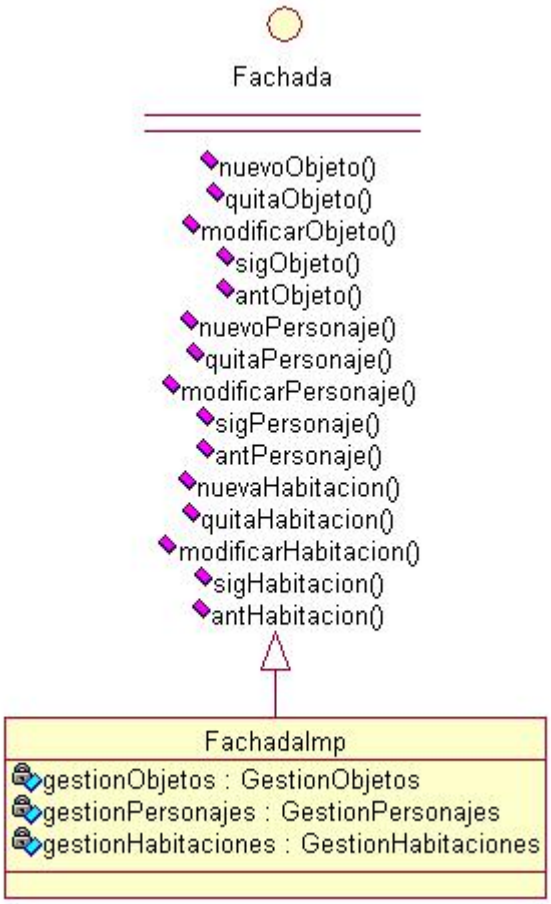
4.06 La clase Misión

4.2.3.1. Paquete modelo.fachada

Como indicabamos en el aparatado *4.2.2 Paquete Controlador*, el Controlador delega en la Fachada para realizar las acciones adecuadas.

La Fachada se encuentra en el paquete modelo.Fachada y consta de una interface Fachada, en la que se definen todos los métodos necesarios para realizar cada uno de los eventos que se pueden enviar al Controlador, y una clase FachadaImp que implementa dicha interface y que contiene como atributos las clases encargadas de la gestión de cada uno de los elementos posibles: GestionObjetos, GestionPersonajes, GestionHabitaciones. Podemos ver el diagrama de clase de la Fachada en la imagen 4.07.

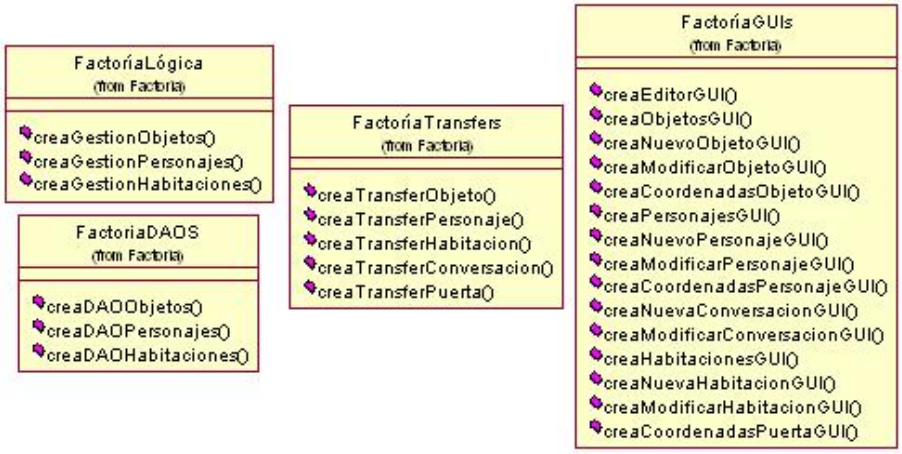
Los datos devueltos por la Fachada serán utilizados por el Controlador para determinar que evento lanza a la interfaz gráfica principal para que se actualice, puesto que en varias ocasiones necesitaremos mostrar un mensaje de error por ejemplo en el caso de intentar modificar un objeto poniéndole el nombre de otro objeto ya existente.



4.07 Diagrama de clase de la Fachada

4.2.3.2. Paquete modelo.factoría

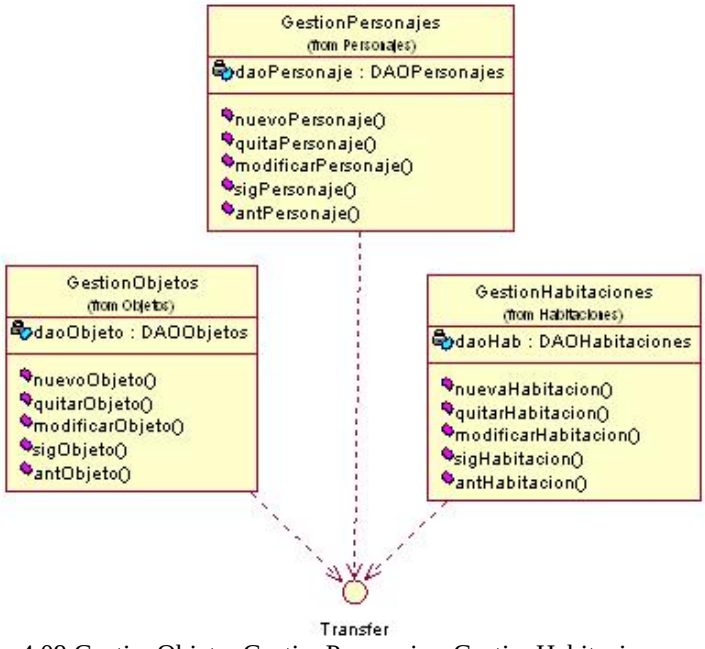
Las Factorías consisten en clases desde las cuales se crean objetos. En este caso disponemos de cuatro tipos de factorías como se aprecia en la imagen 4.08, factorías para la creación de guis (FactoriaGUIs), para la creación de Transfers (FactoríaTransfers), de DAOs (FactoríaDAOs) y para la creación de las clases de gestión de elementos (FactoríaLógica).



4.08 Diagramas de clase de las Factorías

4.2.3.3. Las clases de tipo Gestión

Como se aprecia en la imagen 4.09 las tres clases de gestión de elementos utilizan Transfer ya que a nivel de fachada y gestión no vamos a distinguir entre un tipo de Transfer u otro puesto que para eso tenemos la interface Transfer.



4.09 GestionObjeto, GestionPersonajes, GestionHabitaciones

El paquete modelo también contiene los cinco tipos de transfer que implementan la interfaz Transfer y cuyos atributos pasamos a explicar a continuación, no indicamos las funciones de acceso a los atributos puesto que son triviales:

4.2.3.4. Transfer Objeto

Long id	Id del objeto, coincide con la posición del objeto en el vector de objetos de Misión.
String nombre	Nombre del objeto.
String posX	Coordenada x del objeto en la habitación.
String posY	Coordenada y del objeto en la habitación.
String habitacion	Id de la habitación en la que está el objeto.
String estado1	Nombre del estado inicial del objeto.
String estado2	Nombre del estado secundario del objeto.
String imagenE1	Ruta de la imagen del objeto en el estado inicial.
String imagenE2	Ruta de la imagen del objeto en el estado secundario.
String mirarE1	Texto que se recibe al mirar el objeto en el estado inicial.
String mirarE2	Texto que se recibe al mirar el objeto en el estado secundario.
boolean seCoge	Indica si el objeto se puede coger.
String mensaje	Texto que se recibe al coger el objeto.
String objetoCogido	Id del objeto que se añade al inventario al cogerlo.
String e1SeUsaEn	Nombre del objeto con el que se usa en el estado inicial.
String textoUsar1	Texto que se recibe al usar el objeto en el estado inicial.
String accionE1	Indica que acción se debe realizar al usar el objeto en el estado inicial. Las acciones posibles son cuatro: cambiar el estado del objeto, meter un objeto en el inventario, sacar un objeto del inventario, cambiar al personaje jugador de habitación.

String idDestE1	Indica sobre que objeto se realiza la acción. Tanto para meter objeto en inventario como para cambiar al personaje de habitación se usa el id del objeto y la habitación de destino respectivamente. Para sacar un objeto del inventario y cambiar el estado del objeto se usa el nombre del objeto.
String e2SeUsaEn	Nombre del objeto con el que se usa en el estado secundario.
String textoUsar2	Texto que se recibe al usar el objeto en el estado secundario.
String accionE2	Indica que acción se debe realizar al usar el objeto en el estado secundario. Las mismas acciones que para el estado inicial.
String idDestE2	Indica sobre que objeto se realiza la acción, mismo formato que con el estado inicial.

4.2.3.5. Transfer Personaje

Long id	Id del personaje, indica también su posición en el vector de personajes de Misión.
String nombre	Nombre del personaje.
String imagen	Ruta de la imagen del personaje.
String habitacion	Id de la habitación en la que está el personaje.
String posX	Coordenada x del personaje en la habitación.
String posY	Coordenada y del personaje en la habitación.
String mirar	Texto que se recibe al mirar al personaje.
String objeto1	Nombre del primer objeto que se puede usar con el personaje.
String estado1	Nombre del estado en el que se usa el objeto1 sobre el personaje.
String accion1	Acción que se realizar al usar el objeto1.
String dest1	Objeto sobre el que se realiza la acción.

String conv1	Conversación que se ejecuta al usar el primer objeto sobre el personaje.
String objeto2	Nombre del segundo objeto que se puede usar con el personaje.
String estado2	Nombre del estado en el que se usa el objeto2 sobre el personaje.
String accion2	Acción que se realizar al usar el objeto2.
String dest2	Objeto sobre el que se realiza la acción.
String conv2	Conversación que se ejecuta al usar el segundo objeto sobre el personaje.
Vector conversaciones	Vector con todas las conversaciones del personaje, teniendo en cuenta que la conversación en el índice 0 del vector será la inicial.

4.2.3.6. Transfer Conversación

String nombre	Nombre de la conversación, no se podrá cambiar al modificar una conversación puesto que está ligado a posibles respuestas tanto a opciones de contestación como al uso de objetos sobre personajes.
boolean empiezaJug	Indica si empieza hablando el jugador.
boolean terminaConv	Indica si con esta conversación se finaliza la conversación general.
Vector textoJug	Vector con las frases del jugador en la conversación, cada frase corresponde con un índice del mismo.
Vector textoPj	Vector con las frases del personaje no jugador en la conversación, también una frase por índice del vector.
Vector opciones	Vector con las posibles opciones que se podrán contestar al final de la conversación, máximo cuatro.
String accion	Acción que se realiza al terminar la conversación, solo se produce en conversaciones que cumplan terminaConv. Las acciones posibles son las cuatro que se citaban en el transferObjeto.
String dest	Elemento que sufrirá el efecto de la acción.

4.2.3.7. Transfer Habitación

Long id	Id de la habitación, indica también su posición en el vector de habitaciones de Misión.
String nombre	Nombre de la habitación.
String imagen	Ruta de la imagen de la habitación.
String ancho	Ancho de la habitación.
String alto	Alto de la habitación.
String mascara	Rango de posiciones en las que se podrá colocar al jugador cuando entre en la habitación.
Vector puertas	Vector de transfer puertas que contiene todas las puertas de la habitación.

4.2.3.8. Transfer Puerta

String nombre	Nombre de la puerta
String puertaX	Coordenada x de la posición de la puerta en la habitación.
String puertaY	Coordenada y de la posición de la puerta en la habitación.
String idHabDest	Id de la habitación de destino.
String dX	Coordenada x de la puerta de acceso a la habitación editada desde la habitación adyacente.
String dY	Coordenada y de la puerta de acceso a la habitación editada desde la habitación adyacente.

Con la información almacenada en estos transfers se crearán posteriormente las clases ObjetoEditor y PersonajeEditor a las que se hace referencia en la Introducción, así como las habitaciones que se crearán directamente con los datos del transfer habitación y el vector de transfer puerta que contiene el propio transfer habitación.
Cabe destacar que los cinco tipos de transfers implementan la clase serializable para permitir que se puedan escribir directamente como fichero al grabar las misiones, ya que éstas contienen vectores de transfers.

4.2.4. Paquete integración

En el paquete integración se encuentran únicamente los DAOs relacionados con el subpaquete de cada clase que implementan la interfaceDAO, como se muestra en el diagrama de la imagen 4.10. En el paquete principal integración se encuentra la interfaceDAO que como se indica en la introducción contiene los métodos:

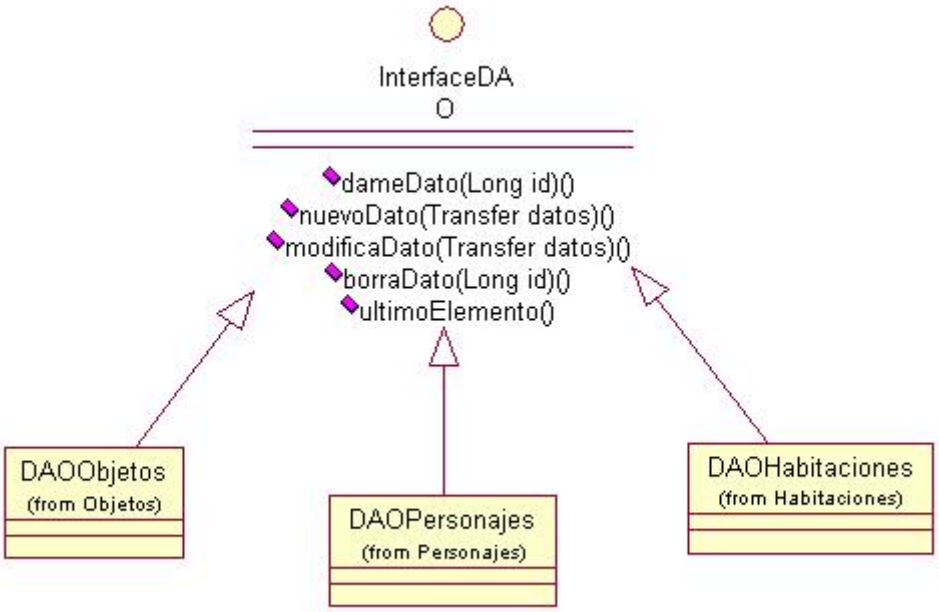
- *Transfer dameDato (Long id) throws IOException;*

- *Long nuevoDato (Transfer datos) throws IOException;*

- *boolean modificaDato (Transfer datos) throws IOException;*

- *void borraDato (Long id) throws FileNotFoundException, IOException;*

- *int ultimoElemento () throws IOException;*



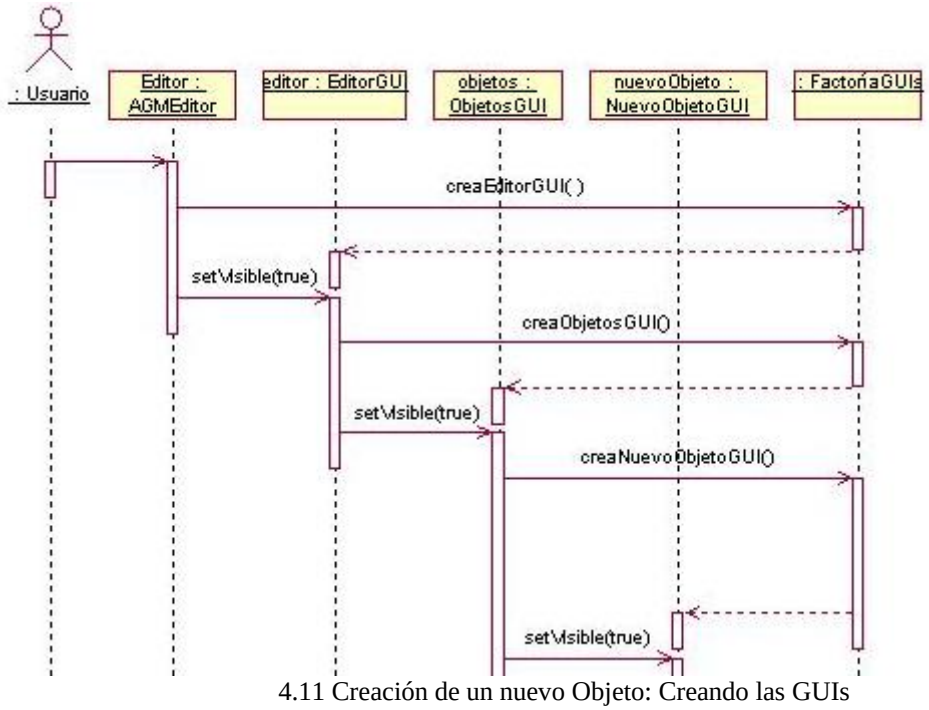
4.10 Diagrama de clases del paquete Integración

4.3. Diagramas de Secuencia

A continuación mostramos algunos diagramas de secuencia de las acciones más importante de la aplicación, se mostrarán en imágenes por partes puesto que son demasiado amplios. Se mostrarán en concreto el diagrama de secuencia de la creación de un nuevo objeto, el de eliminar un personaje, el de modificar una habitación, el de mostrar el siguiente objeto y un ejemplo de captura de coordenadas en imagen. De esta forma quedarán reflejadas todas las acciones posibles a nivel funcional (puesto que las funciones del tipo anteriorElemento son prácticamente iguales a las del tipo siguienteElemento) y el diseño de los tres módulos principales.

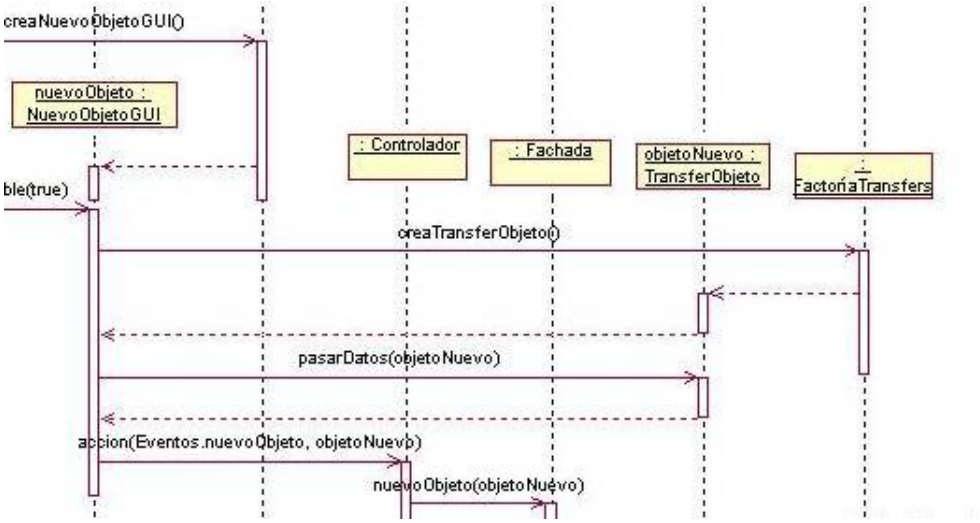
4.3.1. Creación de un nuevo Objeto

Creación de los GUIs desde el principal hasta nuevoObjetoGUI que será en el que introduzcamos los datos, esta parte se obviará en el resto de diagramas de secuencia puesto que no varía en el resto de acciones, salvo el tipo de clase claro. También se obviará en el resto de diagramas el acceso a Factorías para crear los distintos objetos, puesto que queda bien reflejado en los diagramas de nuevoObjeto y sería alargar el resto de diagramas de manera innecesaria.
Como se muestra en el diagrama de la imagen 4.11 desde Editor llamamos a FactoríaGUIs para crear un objeto de tipo EditorGUI, una vez creado lo hacemos visible. Desde editor llamamos nuevamente a FactoríaGUIs para crear un objeto de tipo ObjetosGUI haciendolo nuevamente visible y por último desde objetos repetimos el proceso para crear un objeto de tipo NuevoObjetoGUI



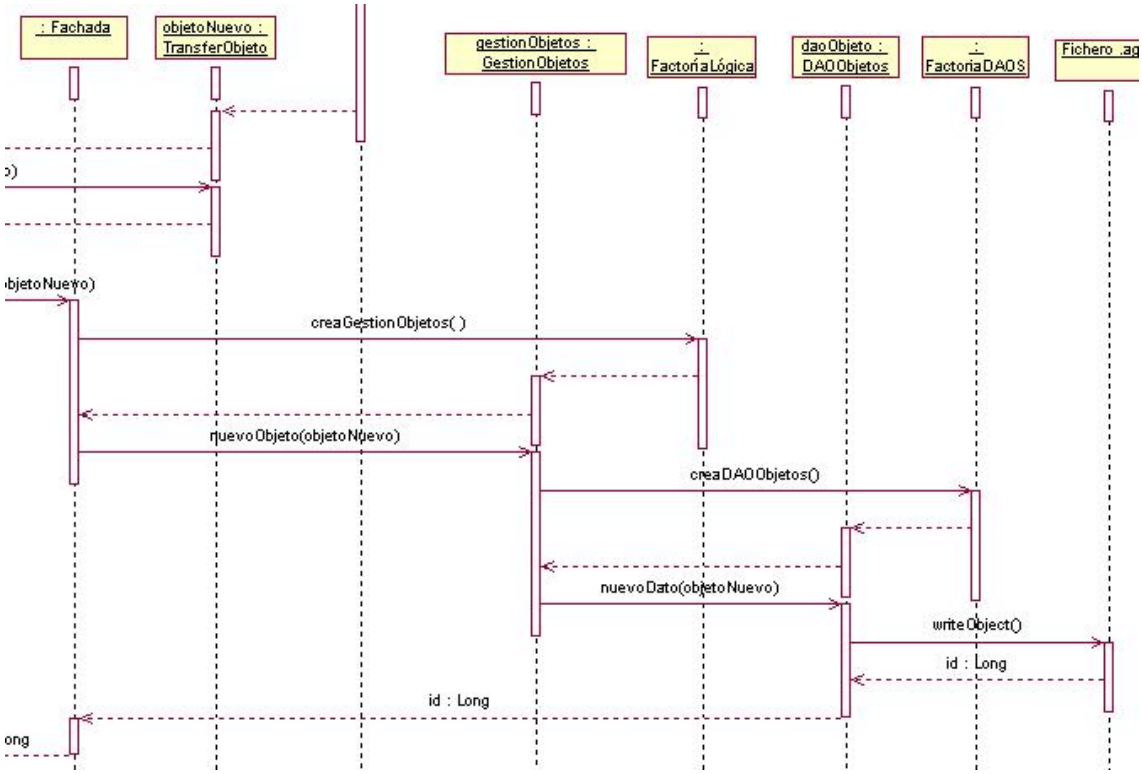
4.11 Creación de un nuevo Objeto: Creando las GUIs

Desde nuevoObjetoGUI, llamamos a FactoriaTransfers para crear un TransferObjeto, y usamos el método pasarDatos (TransferObjeto) de nuevoObjetoGUI para rellenar el transfer con los datos del objeto, una vez tenemos el transfer relleno lanzamos el evento nuevoObjeto al controlador el cual al recibirlo llama al método nuevoObjeto (TransferObjeto) de la fachada, como se aprecia en el diagrama de la imagen 4.12.



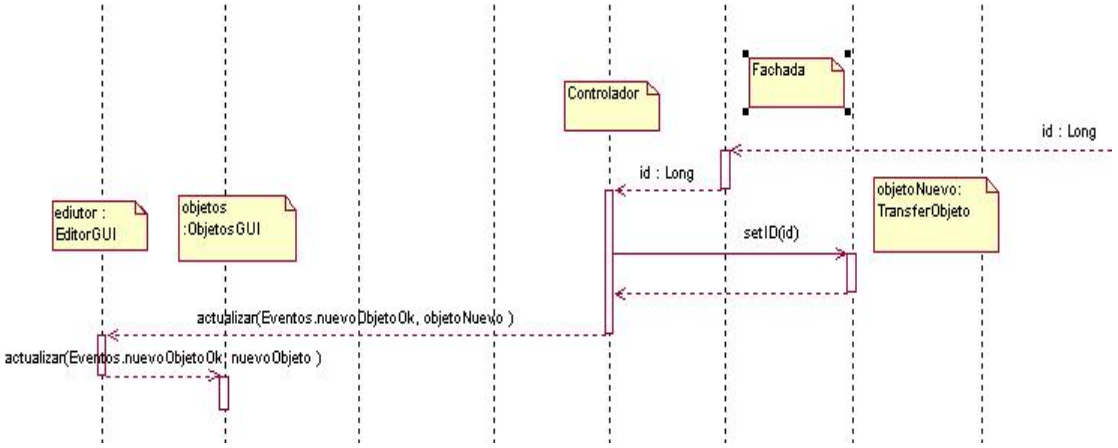
4.12 Creación de un nuevo Objeto: Creando el Transfer

Como se ve en el diagrama de la imagen 4.13, desde la fachada creamos gestionObjetos llamando a la FactoríaLógica, una vez creado se ejecuta nuevoObjeto (TransferObjeto) de gestionObjetos, dentro de gestionObjetos creamos el daoObjeto llamando a FactoríaDAOs y ejecutamos nuevoDato (TransferObjeto) de daoObjeto dentro del cual se escribe finalmente el objeto:



4.13 Creación de un nuevo Objeto: Ejecutando la acción

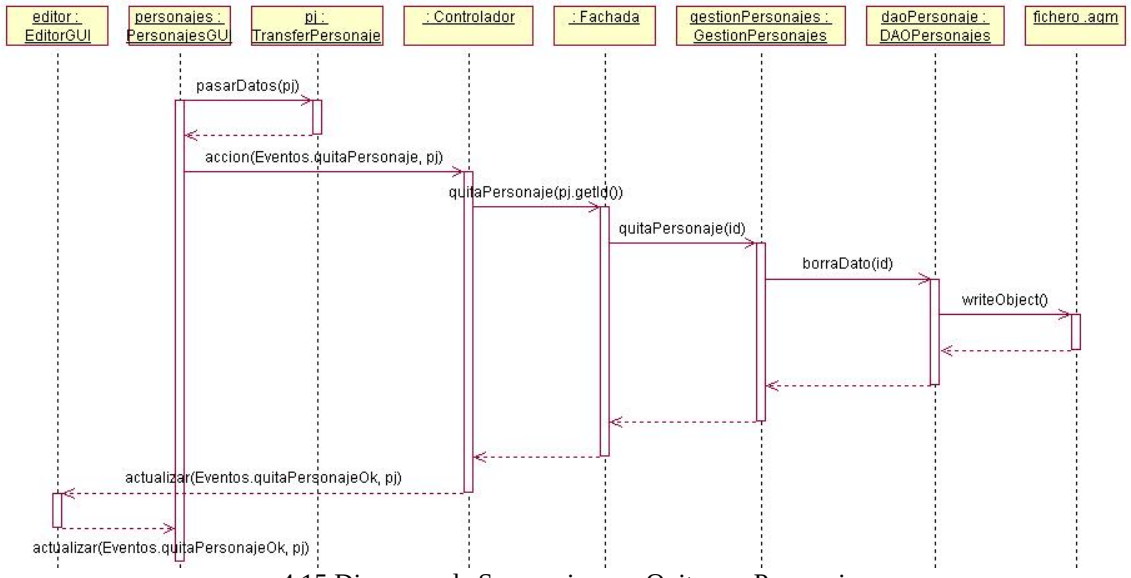
Tras haber escrito el objeto en el fichero se devuelve el id del objeto en formato Long hasta el Controlador, el cual lanza el evento nuevoObjetoOk hasta EditorGUI que a su vez manda el evento hasta ObjetosGUI para que se actualice, como se aprecia en el diagrama de la imagen 4.14.



4.14 Creación de un nuevo Objeto: Actualizando la interfaz

4.3.2. Quitar un personaje

Obtenemos el id del personaje a eliminar desde PersonajesGUI mediante el método pasarDatos() puesto que al pulsar sobre el botón eliminar Personaje la acción se realizará sobre el personajes que se esté mostrando en ese momento.
Una vez tenemos el TransferPersonaje con el id necesario lanzamos el Evento quitaPersonaje al Controlador junto con el transfer del personaje, como se muestra en el diagrama de la imagen 4.15.
En el Controlador cogemos únicamente el id del personaje y ejecutamos el método quitaPersonaje(id) de la Fachada, la cual delegará a su vez en GestionPersonajes que llamará al método borraDato(id) de DAOPersonajes, desde el cual se modificará el vector de personajes del objeto misión del fichero .agm eliminando el personaje del vector que coincida con el id que se le pasa por parámetro.



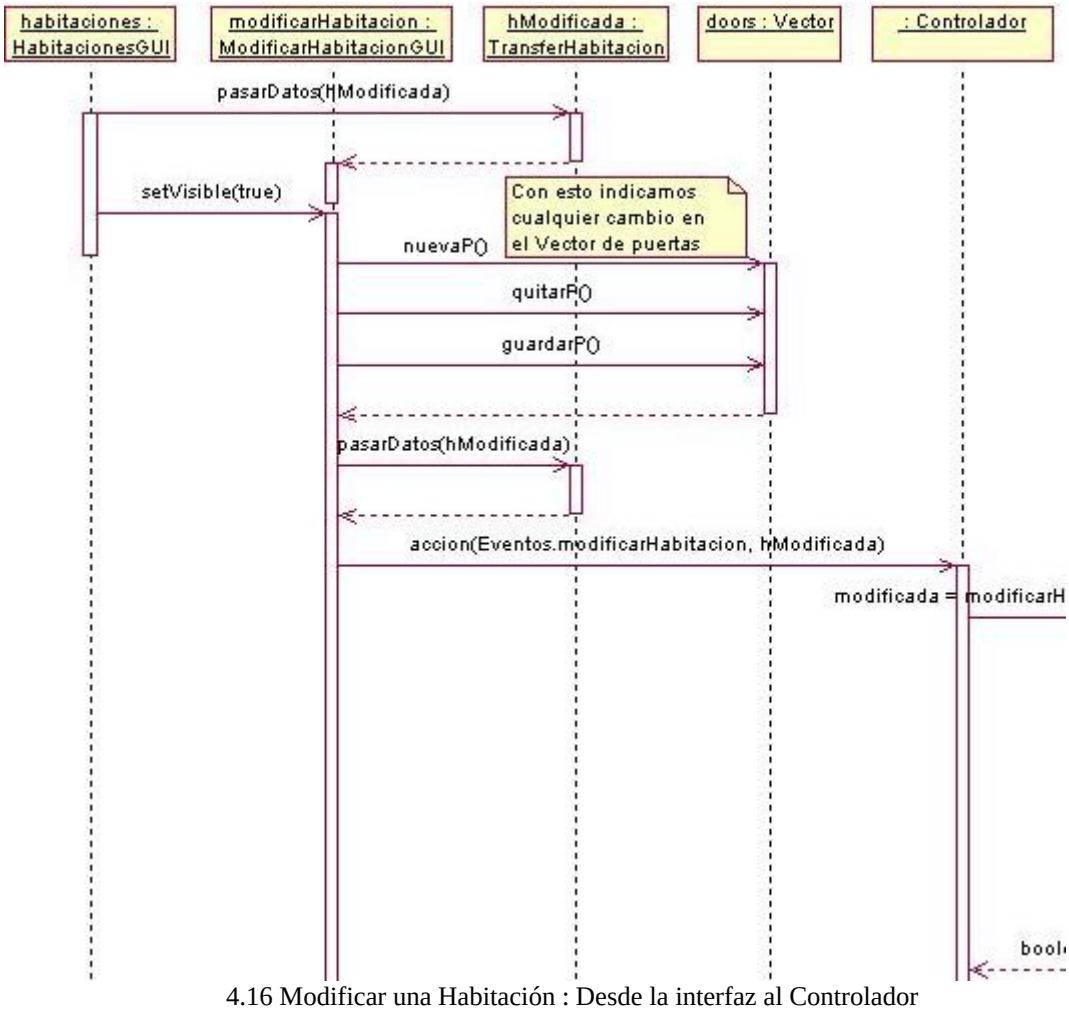
4.15 Diagrama de Secuencia para Quitar un Personaje

4.3.3. Modificar una habitación

En este caso dividimos el diagrama en tres partes para que se pueda apreciar mejor:

La primera parte del diagrama transcurre desde que se realiza una acción en la interfaz correspondiente hasta que llega el evento al Controlador y se corresponde con el diagrama de la imagen 4.16.

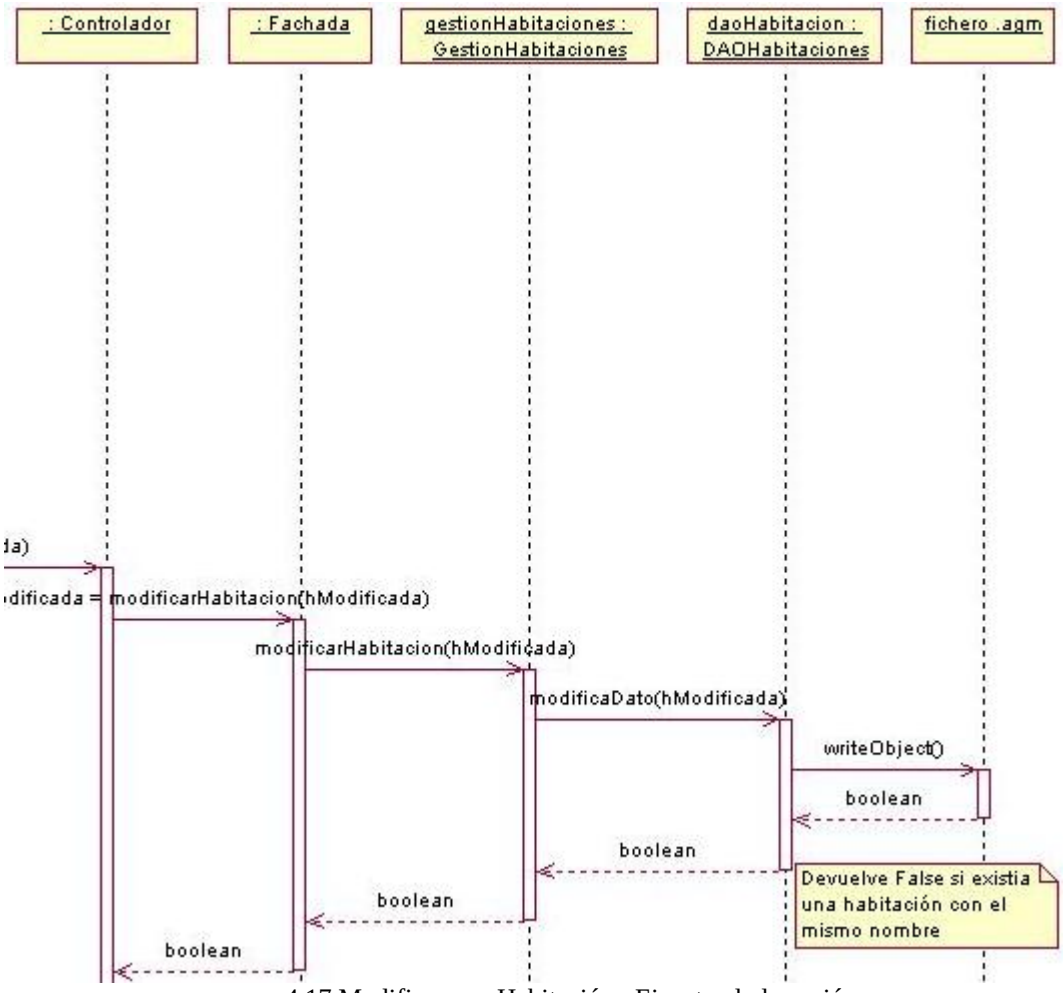
Desde HabitacionesGUI se llama a FactoríaGUIs para crear ModificarHabitaciónGUI y es en ese constructor donde se pasa por parámetro la función pasarDatos (hModificada). No se pone directamente así en el diagrama para evitar poner todas las factorías como se indico en el primer diagrama.
El Vector doors se utiliza para almacenar las puertas de la habitación. En el diagrama ponemos que se ejecutan nuevaP, quitarP y guardarP pero no tienen porque ejecutarse todas, de hecho puede que no se ejecute ninguna, simplemente lo ponemos para indicar que para cuando se actualiza el Transfer hModificada con el método pasarDatos () ya se han hecho todos los cambios en el vector de puertas.



4.16 Modificar una Habitación : Desde la interfaz al Controlador

La segunda parte del diagrama consiste en la ejecución de la acción desde el Controlador hasta que se devuelve el valor al propio Controlador, y se corresponde con el diagrama de la imagen 4.17.

Una vez distinguimos el evento en el controlador llamamos al método modificarHabitación(TransferHabitacion) de la fachada el cual devuelve un booleano indicando si se ha podido modificar la habitación o no, por lo tanto igualamos un booleano a este método para posteriormente poder comprobarlo.
La fachada llama al método del mismo nombre de GestionHabitaciones desde el cual se accede al método modificarDato(TransferHabitacion) de DAOHabitaciones, donde se escribe de nuevo el objeto de tipo misión con los cambios hechos si es que se hicieron y se devuelve el booleano hasta el controlador.



4.17 Modificar una Habitación : Ejecutando la acción

La tercera parte del diagrama consiste en la actualización de la interfaz desde la que se realizó la acción y se corresponde con el diagrama de la imagen 4.18.

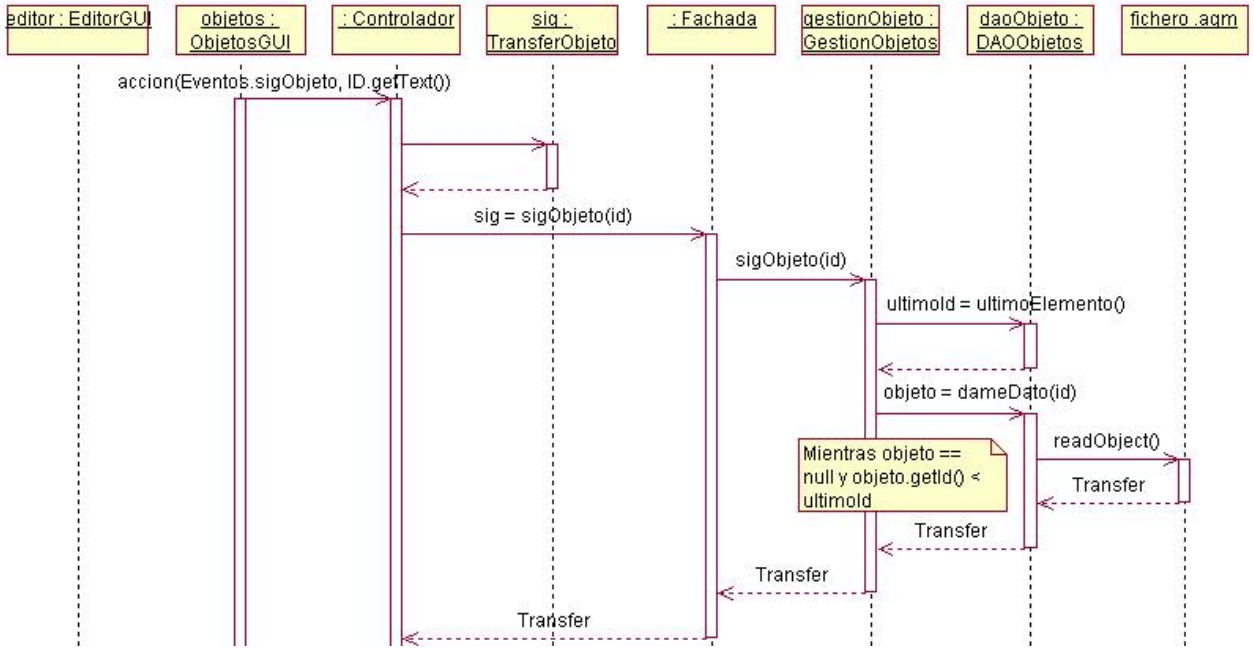
Podemos ver en este caso como el Controlador dependiendo del valor devuelto por la Fachada lanzará un evento u otro a la interfaz inicial, en caso de que se haya modificado la habitación se lanzará detalleHabitación que hará que se muestre en la interfaz principal de las habitaciones los datos de la habitación modificada. Si no se ha modificado la habitación se lanza el evento habitacionNoModifcada y se mostrará un mensaje informando del posible motivo de dicha no modificación.

4.18 Modificar una Habitación : Actualizando la interfaz

4.3.4. Siguiente Objeto

Todas las acciones que suceden hasta que se devuelven los datos al Controlador después de relizar la acción se muestran en el diagrama de la imagen 4.19.
En este caso el dato que necesitamos pasar al Controlador es únicamente el id del siguiente objeto respecto del que se esté mostrando por pantalla cuando se realiza la acción, por lo tanto al pulsar sobre el botón siguiente objeto se lee el id del objeto actual y se le suma uno. Una vez tenemos el id lanzamos el evento sigObjeto junto con el id al Controlador.
Desde el controlador se llama al método sigObjeto(id) y se iguala a un objeto de tipo TransferObjeto que será lo que se nos devuelva, en este caso puede que el objeto sea null puesto que es posible que al pulsar sobre siguiente ya se estuviese mostrando el último objeto, por lo tanto habrá que comprobar posteriormente si el TransferObjeto devuelto es null o no.
La fachada delega en GestionObjeto mediante el método sigObjeto(id) dentro del cual se hacen una serie de comprobaciones.

En primer lugar obtenemos el id del último objeto llamando al método ultimoElemento() de DAOObjeto, sabiendo el id final se hace un bucle mientras que el objeto obtenido sea distinto de null y su id menor o igual a la del ultimo Id de los objetos.
Esto se hace porque cuando eliminamos un objeto no lo quitamos del Vector de objetos del fichero si no que lo ponemos a null, por lo tanto es posible que estemos en el objeto de id 3 y hayamos eliminado (igualado a null) el objeto de id 4 pero si exista un objeto de id 5 por lo tanto al pulsar sobre siguiente de no hacer estas comprobaciones devolvería el objeto 4 que es null y nunca llegaría al objeto de id 5 que es el siguiente objeto real al de id 3.

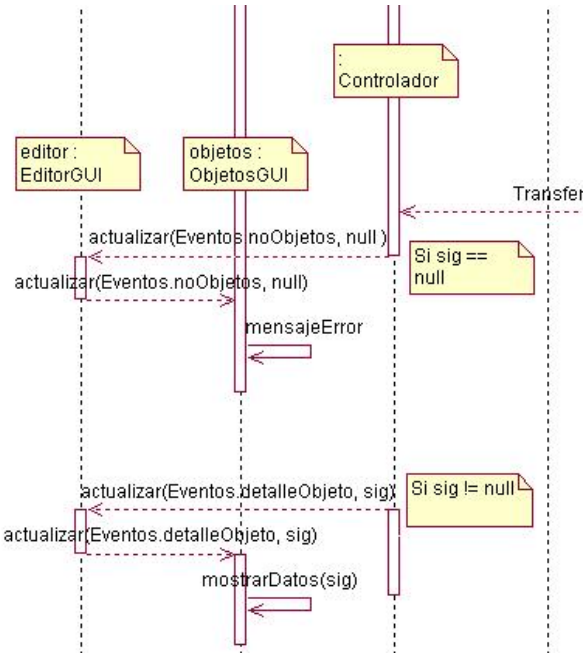


4.19 Siguiente Objeto : Primera Parte

Si termina el bucle y no se ha encontrado un objeto distinto de null quiere decir que no hay más objetos por lo tanto se devuelve null, en caso contrario se devuelve el objeto encontrado.

Una vez devuelto el objeto de tipo TransferObjeto hasta el Controlador hay que comprobar si es null o no, como se muestra en el diagrama de la imagen 4.20. Si el objeto es null se lanza el evento noObjetos a la interfaz principal, si es un objeto normal se lanza el evento detalleObjeto junto con el TransferObjeto devuelto.

Desde la interfaz principal se lanza el evento actualizar a la interfaz desde la que se realizón la acción con el evento correspondiente en cada caso.
Si se recibe el evento noObjeto se muestra un mensaje indicando que no hay más objetos, si se recibe detalleObjeto se muestran los datos del objeto devuelto.



4.20 Siguiente Objeto : Actualización de la intefaz

4.3.5. Captura de coordenadas de un personaje

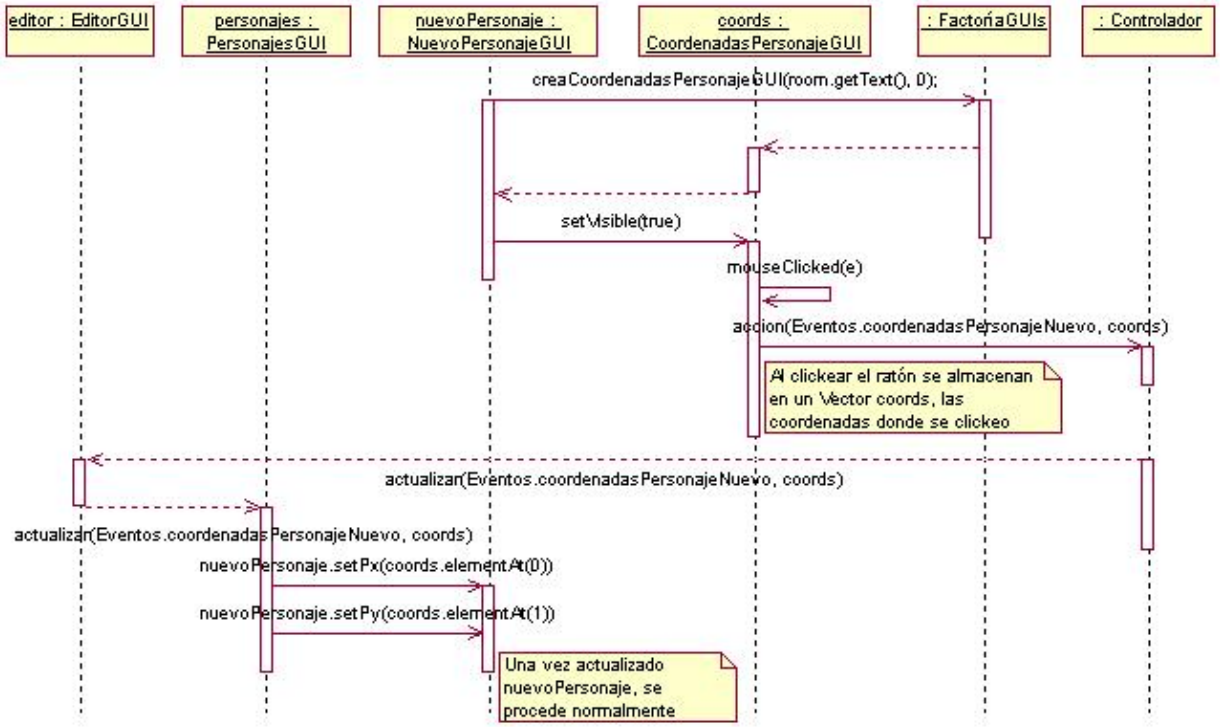
En este caso mostramos la llamada a FactoriaGuis pero solo al crear CoordenadasPersonajeGUI para que se vean los parámetros que se necesitan para que se cree. El diagrama de secuencia correspondiente se puede ver en la imagen 4.21.

Los dos atributos que se pasan por parámetro a la factoríaGUIs para crear CoordenadasPersonajeGUI son un String y un entero, en este caso pasamos room.getText() y 0.
La variable room.getText () corresponde con el campo de texto de nuevoPersonaje donde escribimos el id de la habitación en la que está el personaje, de tal forma que podamos mostrar la imagen en coords usando la ruta general de imágenes de la aplicación más el String “H” concatenado con room.getText ().
El 0 en el constructor será la variable tipo en coords, que nos indica como comentábamos en los diagramas de clases que se ha creado coords desde nuevoPersonaje y no desde modificarPersonaje.

Una vez tenemos creado CoordenadasPuertaGUI se nos mostrará una pantalla con únicamente una imagen de fondo, al clickear sobre la imagen en la posición deseada se lanzará el evento coordenadasPersonajeNuevo (puesto que pasamos un 0 al constructor de CoordenadasPuertaGUI, si hubiesemos pasado un 1 significaría que se creaba desde modificarPersonaje y se lanzaría el evento coordenadasPersonajeMod) al Controlador junto con un Vector en el que se han almacenado en la posición 0 la coordenada x de la posición en la que se clickeó y en la 1 la coordenada de la posición y.

El Controlador al recibir dicho evento lo lanza de nuevo a EditorGUI y desde allí se lanza de nuevo hasta PersonajesGUI. Una vez llega el evento junto con las coordenadas se actualiza nuevoPersonajeGUI modificando directamente los campos de texto Px y Py en los que se indican las coordenadas del personajes en la habitación.

Lo que se consigue de esta forma es tener las coordenadas escritas en sus campos de texto correspondientes para cuando se de a Aceptar para crear el personaje nuevo se añadan al Transfer que se mandará al controlador.



4.21 Diagrama de Secuencia de la captura de coordenadas de un personaje

4.4. Conexión con AGM

Una vez tenemos creado el fichero .agm con el objeto Mision escrito en él el siguiente paso será leer dicho fichero en AGM y crear todos los objetos, personajes y habitaciones que contenga.

Para cargar la misión disponemos de un fichero de texto de nombre mision.txt en el directorio principal del proyecto, en este fichero de texto se escribió la ruta del fichero .agm que contiene la misión que queremos cargar al pulsar en Importar Misión en la interfaz principal del editor de misiones.

La carga del fichero .agm se realiza desde el método CreaMundo() de la clase GestorSistemaBean; una vez tenemos el objeto de tipo misión cargado el primer paso es comprobar si alguno de los objetos, personajes o puertas de nuestra misión se encuentra en las habitaciones por defecto de AGM, esto se debe comprobar mientras se crean cada una de las habitaciones de la aplicación puesto que una vez creadas no se pueden modificar sus atributos. Para comprobarlo utilizamos los métodos objetosEn(), personajesEn() y puertasEn() que devuelven un Vector con todos los objetos, personajes o puertas que se encuentran en la habitación con identificador id:

```
private Vector objetosEn(String id) {
        Vector obs = new Vector();
        Vector objetos = m.getObjetos();
        for (int i = 0; i < objetos.size(); i++)
        {
                TransferObjeto o = (TransferObjeto) objetos.elementAt(i);
                if (new String ("H" + o.getHabitacion()).compareTo(id) == 0)
                        obs.add(o);
        }
        return obs;
}
```

- m es el objeto de tipo misión.

```
private Vector personajesEn (String id) {
        Vector obs = new Vector();
        Vector personajes = m.getPersonajes();
        for (int i = 0; i < personajes.size(); i++)
        {
                TransferPersonaje p = (TransferPersonaje) personajes.elementAt(i);
                if (new String ("H" + p.getHabitacion()).compareTo(id) == 0)
                        obs.add(p);
        }
        return obs;
}
```

```
private Vector puertasEn(String id) {
        Vector obs = new Vector();
        Vector habitaciones = m.getHabitaciones();
        for (int i = 0; i < habitaciones.size(); i++)
        {
        TransferHabitacion h = (TransferHabitacion) habitaciones.elementAt(i);
        Vector puertas = h.getPuertas();
        for (int j = 0; j < puertas.size(); j++)
             if (new String("H" + ((TransferPuerta)
              puertas.elementAt(j)).getHDest()).compareTo(id) == 0)
                obs.add(puertas.elementAt(j));
        }
        return obs;
}
```

También usaremos objetosEn() y personajesEn() para meter los objetos y personajes que se encuentren en las habitaciones de la misión editada.
Una vez tenemos todos los objetos, personajes y puertas de una habitación pasamos a crearlos. Las puertas se crearán directamente puesto que no necesitan de una clase para controlar su comportamiento, sin embargo para los objetos y personajes usaremos las clases ObjetoEditor y PersonajeEditor respectivamente, que como se comenta en la introducción serán comunes para todos los objetos y personajes editados.

Para crear las habitaciones editadas recorremos el vector de habitaciones de la misión y para cada habitación distina de null creamos las puertas y posteriormente comprobamos si hay objetos o personajes en dichas habitaciones con los métodos indicados anteriormente, en caso de haberlos se crean:

```
private void crearEditados(){
try
{
Vector habitaciones = m.getHabitaciones();
for (int i = 0; i < habitaciones.size(); i++)
{
	TransferHabitacion h = (TransferHabitacion)habitaciones.elementAt(i);
	if (h != null)
	{
		String idHab = "H" + h.getId();
		LinkedList puertas = new LinkedList();
		LinkedList objetos = new LinkedList();
		LinkedList pjs = new LinkedList();
		String idO;
		String idPNJ;
		Vector doors = h.getPuertas();
		for (int j = 0; j < doors.size(); j++)
		{
			idO = "O"+ contObjetos++;
			TransferPuerta puerta = (TransferPuerta)doors.elementAt(j);
			puertas.add(new Puerta(idO, idHab, "H" + puerta.getHDest(),
				Integer.parseInt(puerta.getPX()),
				Integer.parseInt(puerta.getPY()), "ABIERTA"));
		}
		Vector objs = objetosEn(idHab);
		for (int j = 0; j < objs.size(); j++)
		{
			idO = "O"+ contObjetos++;
			TransferObjeto o = (TransferObjeto) objs.elementAt(j);
			ObjetoEditor ob = new ObjetoEditor(o, idO, m.getObjetos());
			ob.setPJs(new LinkedList());
			objetos.add(ob);
		}
		Vector pnjs = personajesEn(idHab);
		for (int j = 0; j < pnjs.size(); j++)
		{
			idPNJ ="PNJ"+ contPNJ++;
			TransferPersonaje p = (TransferPersonaje) pnjs.elementAt(j);
			pjs.add(new PersonajeEditor(p, idPNJ, m.getObjetos()));
		}
		crearHabitacion(idHab,h.getNombre(),true, Integer.parseInt(h.getAlto()),
		Integer.parseInt(h.getAncho()), h.getMascara(), puertas, objetos, pjs);
		System.out.println(h.getNombre() + " " + idHab + " creado.");
	}
}
}
catch (Exception e){
	System.out.println("Error al cargar el fichero de misiones");
	e.printStackTrace();
}
}
```

4.4.1. La clase ObjetoEditor

La clase ObjetoEditor extiende a la clase ObjetoNoPersonajeClase, consiste en una ampliación de ésta en la que se definen todas las acciones comunes a los objetos, tales como comprobación de la acción que se ha pulsado o control del cambio de estado, distinguiendo las acciones que se devuelven según los datos almacenados en el transferObjeto que se pasa por el constructor y que se rellenó en la interfaz del Editor. A continuación mostramos el código fuente de ObjetoEditor por partes para facilitar su entendimiento:

```
public class ObjetoEditor extends ObjetoNoPersonajeClase {

private LinkedList PJs;

private String nombre;
private String estado1;
private String estado2;
private String mirarE1;
private String mirarE2;

private ObjetoEditor objetoCogido;
private String e1SeUsaEn;
private String textoUsar1;
private String accionE1;
private String destE1;
private String e2SeUsaEn;
private String textoUsar2;
private String accionE2;
private String destE2;
private ObjetoEditor cogidoE1;
private ObjetoEditor cogidoE2;
```

Las variables de la clase ObjetoEditor como se puede apreciar coinciden la mayoría con las de TransferObjeto puesto que básicamente creamos el objeto a partir del Transfer por lo tanto su utilidad se puede ver en el aparatado *4.2.3.4 Transfer Objeto*.

Hay que destacar las variables objetoCogido, cogidoE1 y cogidoE2, estos tres objetos son de tipo ObjetoEditor.
El objeto objetoCogido será el que se introduzca en el inventario al usar la acción coger sobre el objeto principal, si no se puede coger será null. Los objetos cogidoE1 y cogidoE2 serán los objetos que se meten en el inventario si al realizar la acción Usar como resultado de la misma se introduce un objeto en el inventario, cogidoE1 será el que se introduzca si se produce en el estado inicial de objeto y cogidoE2 si es en el estado secundario.
Por último el atributo de tipo LinkedList PJs nos servirá para saber si un objeto ha sido ya cogido por un personaje, de tal forma que no se pueda coger más veces.

Al constructor se le pasa por parámetro un TransferObjeto, un String idO con el id asignado al objeto en GestorSistema y un Vector con todos los objetos editados para en caso de que se pueda coger un objeto poder acceder a sus datos.
En primer lugar se llama al constructor de objetoNoPersonajeClase con los datos básicos del objeto y después se da nombre al tipo con el paquete en el que está la clase seguido del nombre del objeto, ya que todos los objetos editados son del mismo tipo de clase se incluye el nombre en el tipo para poder distinguirlos.

```
public ObjetoEditor(TransferObjeto ob, String idO, Vector objetos){
    super(idO, ob.getEstado1(), ob.getHabitacion(), Integer.parseInt(ob.getPosX()),
        Integer.parseInt(ob.getPosY()));
    tipo = "agm.objetos.objetosNoPersonaje." + ob.getNombre();
    nombre = ob.getNombre();
    estado1 = ob.getEstado1(); estado2 = ob.getEstado2();
    mirarE1 = ob.getMirarE1(); mirarE2 = ob.getMirarE2();
    seCoge = ob.isSeCoge();
    mensaje = ob.getMensaje();
    if (seCoge && objetos != null)
    {
        TransferObjeto cogido =
        (TransferObjeto)objetos.elementAt(Integer.parseInt(ob.getObjetoCogido()));
        cogido.setPosX("0"); cogido.setPosY("0");
        objetoCogido = new ObjetoEditor(cogido, null, null);
    }
    else objetoCogido = null;
    e1SeUsaEn =  ob.getE1SeUsaEn();
    e2SeUsaEn = ob.getE2SeUsaEn();
    textoUsar1 = ob.getTextoUsar1();
    textoUsar2 = ob.getTextoUsar2();
    accionE1 = ob.getAccionE1();
    accionE2 = ob.getAccionE2();
    destE1 = ob.getIdDestE1();
    destE2 = ob.getIdDestE2();
    if (accionE1.compareTo("Meter objeto en el inventario") == 0 && objetos != null)
    {
        TransferObjeto cogido =
        (TransferObjeto)objetos.elementAt(Integer.parseInt(ob.getIdDestE1()));
       cogidoE1 = new ObjetoEditor(cogido, null, null);
    }
    if (accionE2.compareTo("Meter objeto en el inventario") == 0 && objetos != null)
    {
        TransferObjeto cogido =
        (TransferObjeto)objetos.elementAt(Integer.parseInt(ob.getIdDestE2()));
        cogidoE2 = new ObjetoEditor(cogido, null, null);
    }
}
```

El resto de asignaciones son triviales por lo que pasamos a explicar los bloques If.
El primer bloque If nos sirve para saber si se puede coger un objeto al realizar la acción coger sobre el objeto editado, en caso afirmativo cogemos el id del objeto que está almacenado en el transfer y accedemos a la posición del vector de objetos el índice del objeto deseado en el vector coincidirá con el id devuelto por el transfer, por lo tanto accedemos directamente a él.
Una vez tenemos el TransferObjeto del objeto que se cogerá ponemos las variables posX y posY de dicho Transfer a 0, esto lo hacemos porque al llamar por medio de super al constructor de objetoNoPersonajeClase hay que pasarle la posición del objeto, y al tratarse de objetos creados para ser metidos directamente al inventario sus variables posX y posY serán vacías por lo tanto saltaría excepción al intentar pasar a entero dichas variables. Por ultimo creamos el ObjetoEditor pasando como parámetros el Transfer obtenido y modificado, null en el IdO puesto que se le asigna automáticamente un id al meterse en el inventario del personaje y con el vector de objetos también a null porque ya es un objeto cogido por lo tanto no se podrá coger de nuevo.

De la misma forma que creamos el objeto que se cogerá al realizar la acción coger generamos también los dos posibles objetos que se meterán al inventario al usarse el objeto tanto en el estado inicial como en el secundario.

```
public void cambiaEstado()
{
    if (estado.equals(estado1))
       estado = estado2;
    else
       estado = estado1;
}

public void setPJs(LinkedList lista)
{
    PJs = lista;
    guardarLista();
}
```

El método cambiaEstado() es trivial, simplemente pasa de un estado a otro. El método setPJs servirá para pasar una LinkedList al atributo Pjs del objeto y posteriormente guardar la lista mediante el método guardarLista(), se podría eliminar este método y hacerlo directamente en el constructor pero hemos preferido mantenerlo por no modificar la estructura del resto de objetos no editados.

A continuación explicamos el método ejecutarAccion dividido por cada acción que se puede realizar sobre el objeto:

```
public Acciones ejecutarAccion(String idA, String IdOd, String IdPj) {
try {
  if ("MIRAR".equals(idA)) {
     if (estado.equals(estado1)) {
         return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
               Habitacion h = conseguirHabitacion(idHab);
               ObjetoActualizacion oA = generaNotificacion(idPJ, mirarE1);
               mostrarResultado(h, oA);
            }

             public boolean interpretaCondicion(Habitacion hab1,
                              Habitacion hab2, String idPJ) {
                 return true;
             }
          };
     }
     else
        return new Acciones() {
           public void ejecutar(String idHab, String idPJ) {
              Habitacion h = conseguirHabitacion(idHab);
              ObjetoActualizacion oA = generaNotificacion(idPJ, mirarE2);
              mostrarResultado(h, oA);
           }

            public boolean interpretaCondicion(Habitacion hab1,
                              Habitacion hab2, String idPJ) {
                return true;
             }
        };
  }
```

La acción Mirar es bastante simple, comprobamos en que estado está el objeto cuando se realiza la acción y dependiendo del estado en el que esté generamos la notificación con la variable mirarE1 o mirarE2 para el estado inicial y el secundario respectivamente.

```
else if ("COGER".equals(idA)) {
  if (seCoge) {
     if (!(PJs.contains(IdPj + this.tipo))) {
         PJs.add(IdPj + this.tipo);
         guardarLista();
         return new Acciones() {
             public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                meterEnInventario(h, idPJ,
"agm.objetos.objetosNoPersonaje.ObjetoEditor", objetoCogido, mensaje);
             }

             public boolean interpretaCondicion(Habitacion hab1,
                              Habitacion hab2, String idPJ) {
                  return true;
             }
         };
     }
     else
         return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                 Habitacion h = conseguirHabitacion(idHab);
                 ObjetoActualizacion oA = generaNotificacion(
                          idPJ,"Con uno tengo más que suficiente");
                 mostrarResultado(h, oA);
            }

            public boolean interpretaCondicion(Habitacion hab1,
                              Habitacion hab2, String idPJ) {
                 return true;
            }
         };
  }
  else
      return new Acciones() {
          public void ejecutar(String idHab, String idPJ) {
               Habitacion h = conseguirHabitacion(idHab);
               ObjetoActualizacion oA = generaNotificacion(
                                 idPJ, "No puedo coger eso");
               mostrarResultado(h, oA);
          }

          public boolean interpretaCondicion(Habitacion hab1,
                              Habitacion hab2, String idPJ) {
               return true;
          }
      };
}
```

En la acción Coger lo primero que comprobamos es si el objeto se puede coger, si no se puede coger generamos la notificación *"No puedo coger eso"* y mostramos el resultado, como se aprecia en el último parrafo.
Si el objeto se puede coger pasamos a comprobar si el personaje que realiza la acción ya ha cogido el objeto con anterioridad para ello comprobamos si la lista PJs contiene el id del Pj junto con el tipo del objeto, no basta con poner el nombre del PJ puesto que todos los objetos editados son de la misma clase, esto lo explicamos más a fondo posteriormente en el apartado *4.6. Problemas encontrados durante el desarrollo.* Si el personaje ya posee el objeto generamos la notificación "*Con uno tengo más que suficiente"* y mostramos el resultado.
Si el objeto se puede coger y el personaje no lo tiene ya, añadimos el id del pj concatenado con el tipo del objeto en la lista de PJs para que no lo pueda coger de nuevo más tarde. Una vez hecho esto, realizamos la acción meterEnInventario en la que introducimos por parámetro el id de la habitación en la que está el objeto, el id del personaje, la clase del objeto, el objeto que se cogerá que se creó como indicabamos en el constructor y el mensaje que se recibirá al coger el objeto.

La acción Usar tendrá dos partes prácticamente iguales, Usar si estamos en el estado inicial y Usar si estamos en el estado secundario. Lo primero que hacemos es buscar en la habitación en la que está el personaje el objeto que coincide con el IdOd que es sobre el que se realizó la acción, una vez lo tenemos se guarda en oNPC y su nombre en nom. A continuación hay que comprobar si el nombre del objeto coincide con el que se usa con el objeto editado tanto en el estado inicial (e1SeUsaEn) como en el secundario (e2SeUsaEn) y si el estado actual del objeto editado coincide con el estado en el que se usa con dicho objeto estado1 para el inicial y estado2 para el secundario.

```
else if ("USAR".equals(idA)) {
   ObjetoNoPersonajeClase oNPC = null;
   HabitacionHome habHome = Acciones.getHabitacionHome();
   Habitacion h = habHome.findByIdPJ(IdPj);
   String nom = null;
   if (IdOd != null) {
     oNPC = h.buscarObjeto(IdOd, IdPj);
     if (oNPC != null)
       nom = ((ObjetoEditor) oNPC).getNombre();
     else
       nom = "";
   }
   else
      nom = "";
   if ((e1SeUsaEn.equals(nom)) && (estado.equals(estado1))) {
      Acciones a = new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
           Habitacion h = conseguirHabitacion(idHab);
           if (accionE1.compareTo("Cambiar el estado del objeto") == 0)
               cambiarEstadoObjeto(h, destE1+idPJ, idPJ, textoUsar1);
           else
            if (accionE1.compareTo("Meter objeto en el inventario") ==0)
              meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.ObjetoEditor",
                      cogidoE1, textoUsar1);
           else if (accionE1.compareTo("Sacar objeto del inventario") == 0)
               sacarDeInventario(h, destE1+idPJ, idPJ, textoUsar1);
           else {
                Habitacion hDest = conseguirHabitacion("H" + destE1);
                cambioHabitacion(h, hDest, idPJ);
                ObjetosRefresco oR = generaRefresco(hDest, idPJ);
                mostrarResultado(h, oR);
           }
        }

        public boolean interpretaCondicion(Habitacion hab1,
                            Habitacion hab2, String idPJ) {
           return true;
        }
      };
      LinkedList parametros = new LinkedList();
      parametros.add(getIdO());
      a.setParametros(parametros);
      return a;
   }
```

Si se cumplen las condiciones `((e1SeUsaEn.equals(nom)) && (estado.equals(estado1)))` quiere decir que el objeto se va a usar en el estado inicial y sobre el objeto oNPC de nombre nom. Por lo tanto, pasamos a comprobar que acción ha de realizarse en ese caso mediante Ifs sobre accionE1 en todas las acciones mostramos como texto el contenido en la variable textoUsar1.
Las posibles acciones a realizar son cambiar el estado del objeto, para la cual introducimos entre otros datos destE1+idPJ, que será el nombre del objeto concatenado con el id del personaje que lo posee, ya que los id asignados a los objetos en los inventarios se hacen con ese formato. Otra acción será meter objeto en inventario, ya explicada en la parte de la acción Coger.

También podremos sacar un objeto del inventario, en la que usaremos el mismo formato para conseguir el objeto a sacar que en cambiar el estado del objeto, es decir, destE1+idPJ. Y por último, podremos cambiar al personaje de habitación, para ello obtenemos la habitación con "H" + destE1, ejecutamos el método cambioHabitacion sobre dicha habitación y generamos un objeto de tipo refresco para actualizar la pantalla.
El último paso en la acción usar para el estado inicial será asignar los parámetros correspondiente a la acción a y devolverla.

Como comentabamos anteriormente el código para Usar en el estado secundario es prácticamente idéntico al de Usar para el estado inicial salvo que en este caso, por supuesto, usamos las variables relacionados con el estado secundario: e2SeUsanEn para comprobar el nombre del objeto, estado2 para comprobar si es el estado en el que se usa el objeto editado con el objeto de nombre nom, textoUsar2 para notificar de la acción realizada y destE2 donde estará el id o nombre del elemento involucrado en la acción:

```
else if ((e2SeUsaEn.equals(nom)) && (estado.equals(estado2))) {
    Acciones a = new Acciones() {
       public void ejecutar(String idHab, String idPJ) {
           Habitacion h = conseguirHabitacion(idHab);
           if (accionE2.compareTo("Cambiar el estado del objeto") == 0)
               cambiarEstadoObjeto(h, destE1+idPJ, idPJ, textoUsar2);
           else if (accionE2.compareTo("Meter objeto en el inventario") == 0)
               meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.ObjetoEditor",
                       cogidoE2, textoUsar2);
           else if (accionE2.compareTo("Sacar objeto del inventario") == 0)
               sacarDeInventario(h, destE2+idPJ, idPJ, textoUsar2);
           else {
                Habitacion hDest = conseguirHabitacion("H" + destE2);
               cambioHabitacion(h, hDest, idPJ);
               ObjetosRefresco oR = generaRefresco(hDest, idPJ);
               mostrarResultado(h, oR);
           }
       }

       public boolean interpretaCondicion(Habitacion hab1,
                           Habitacion hab2, String idPJ) {
             return true;
       }
    };
    LinkedList parametros = new LinkedList();
    parametros.add(getIdO());
    a.setParametros(parametros);
    return a;
}
return new Acciones() {
   public void ejecutar(String idHab, String idPJ) {
      Habitacion h = conseguirHabitacion(idHab);
      ObjetoActualizacion oA = generaNotificacion(idPJ,
            "No puedo hacer eso");
      mostrarResultado(h, oA);
   }

   public boolean interpretaCondicion(Habitacion hab1,
                           Habitacion hab2, String idPJ) {
      return true;
   }
};
}
```

En la última parte del código podemos apreciar como devolvemos la notificación *"No puedo hacer eso"* si no se cumplen las condiciones ni para el estado inicial ni para el secundario.

El último caso que nos queda en el método ejecutarAccion es aquel en el que no se realiza ningún acción posible con el objeto editado, como por ejemplo si pulsasemos Hablar sobre el objeto, para estos casos devolvemos una acción con la notificación "*Nah… Eso no serviría para nada"*:

```
else
   return new Acciones() {
      public void ejecutar(String idHab, String idPJ) {
         Habitacion h = conseguirHabitacion(idHab);
         ObjetoActualizacion oA = generaNotificacion(idPJ,
                   "Nah... Eso no serviría para nada");
         mostrarResultado(h, oA);
      }

      public boolean interpretaCondicion(Habitacion hab1,
                           Habitacion hab2, String idPJ) {
         return true;
      }
    };
} catch (Exception e) {
      return null;
  }

}
```

El último método de ObjetoEditor es guardarLista() que como indicabamos antes se ejecuta desde setPjs() y servirá para almacenar que personajes tienen ya el objeto:

```
public void guardarLista()
{
  try
  {
     DocumentBuilder builder = DocumentBuilderFactory.newInstance()
                   .newDocumentBuilder();
     Document doc = builder.newDocument();

     Element personajes = doc.createElement("Personajes");
     doc.appendChild(personajes);
     int longitud = PJs.size();
     for (int i = 0; i < longitud; i++) {
         Element personaje = doc.createElement("Personaje");
         personajes.appendChild(personaje);
         personaje.appendChild(doc.createTextNode((String) PJs.get(i)));
     }

     Transformer trans = TransformerFactory.newInstance()
                    .newTransformer();
     trans.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
     trans.setOutputProperty(OutputKeys.INDENT, "yes");
     trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
                             "personajes.dtd");
     trans.transform(new DOMSource(doc), new StreamResult(new File(
                    getIdO() + ".xml")));
  }
  catch (Exception e) {
      System.out.println("Error al guardar la lista de personajes");
      e.printStackTrace();
  }
}
```

El método genera un fichero de tipo .xml con nombre igual al idO del objeto de la clase y escribe en el todos los personajes que están contenidos en el LinkedList PJs en el que se han ido guardando según cogían el objeto.

4.4.2. La clase PersonajeEditor

La clase PersonajeEditor extiende a la clase PersonajeNoJugadorClase, consiste en una ampliación de ésta en la que se definen todas las acciones comunes a los personajes, tales como comprobación de la acción que se ha pulsado, teRespondo() o control del cambio de estado, distinguiendo las acciones que se devuelven según los datos almacenados en el transferPersonaje que se pasa por el constructor y que se rellenó en la interfaz del Editor. A continuación mostramos el código fuente de PersonajeEditor dividido en partes para facilitar su entendimiento.

En primer lugar mostramos las variables de PersonajeEditor:

```
public class PersonajeEditor extends PersonajeNoJugadorClase{

private String mirar;
private String ob1;
private String estado1;
private String act1;
private String dest1;
private TransferConversacion conv1;
private String ob2;
private String estado2;
private String act2;
private String dest2;
private TransferConversacion conv2;
private Vector convs;
private TransferConversacion actual;
private Vector objetos;
private boolean usado1;
private boolean usado2;
```

La mayor parte de los atributos son los mismos que los explicados en el aparatado *4.2.3.5 Transfer Personaje* ya que los personajes editados se crean básicamente a partir de los transferPersonaje contenidos en el vector de personajes del objeto de tipo Misión. En este caso conv1 y conv2 son del tipo TransferConversacion a diferencia de en el TransferPersonaje que eran de tipo String, estos dos objetos se crearán en el constructor gracias al String con el nombre de la conversación correspondiente almacenado en el transfer del personaje a crear.

También disponemos del TransferConversacion actual que estará apuntando a la conversación que se está desarrollando actualmente, comenzando por la correspondiente al índice 0 del vector convs en el que están todas las conversaciones del personaje.

Contamos también con un vector objetos en el que estarán todos los objetos editados por si necesitamos acceder a alguno en una determinada acción.
Y, por último, contamos con dos variables booleanas usado1 y usado2 que nos servirán para saber dentro de una conversación si hemos llegado a ella como resultado de usar un objeto sobre el personaje editado, el objeto1 para usado1 y el objeto2 para usado2, de tal forma que podamos realizar las acciones correspondientes indicadas por act1 y act2 respectivamente.

Al constructor se le pasan por parámetro un TransferPersonajes con los datos del personaje editado a crear, un String con el id del personaje asignado en GestorSistema y un Vector con todos los objetos editados de la misión.
Lo primero que hacemos en el constructor es llamar al constructor de PersonajeNoJugadorClase mediante super() al cual le pasamos los datos generales del personaje editado, y en el que indicamos que el estado inicial del personajes sera "*Esperando"*, el cual seguirá así hasta que algún jugador inicie una conversación con el personaje, en ese momento el estado del personajes cambiará al de *"Conversando"*.

```
public PersonajeEditor(TransferPersonaje p, String idPj, Vector obs){
        super(idPj, p.getNombre(), p.getHabitacion(), Integer.parseInt(p.getPosX()),
                Integer.parseInt(p.getPosY()), "ESPERANDO");
         tipo = "agm.objetos.personajesNoJugadores." + p.getNombre();
         mirar = p.getMirar();
         ob1 = p.getObjeto1(); ob2 = p.getObjeto2();
         estado1 = p.getEstado1(); estado2 = p.getEstado2();
         act1 = p.getAccion1(); act2 = p.getAccion2();
         dest1 = p.getDest1(); dest2 = p.getDest2();
         convs = p.getConversaciones();
         for (int i = 0; i < convs.size(); i++)
            if (convs.elementAt(i) != null)
            {
                TransferConversacion c = (TransferConversacion) convs.elementAt(i);
                if (c.getNombre().compareTo(p.getConv1()) == 0)
                        conv1 = c;
                if (c.getNombre().compareTo(p.getConv2()) == 0)
                        conv2 = c;
             }
         objetos = obs;
         actual = (TransferConversacion) convs.elementAt(0);
         usado1 = false; usado2 = false;
  }
```

Al igual que con los objetos editados igualamos el tipo del personajes al paquete en el que se encuentra la clase concatenado con el nombre del personaje.
Igualamos el vector convs al vector de conversaciones del TransferPersonaje y recorremos todas las conversaciones buscando las que coincidan (si es que las hay) con el nombre indicado en Conv1 y Conv2 del transfer del personaje, de tal forma que podamos crear estas conversaciones para poder acceder de manera directa a ellas posteriormente.
Iniciamos las variables usado1 y usado2 a false puesto que recién usado el personaje no se ha usado ningún objeto sobre él.
El resto de asignaciones del constructor son triviales.

```
public void cambiaEstado() {
         if (estado.equals("CONVERSANDO")) {
             estado = "ESPERANDO";
         } else {
             estado = "CONVERSANDO";
         }
}
```

El método cambiaEstado() es trivial, si el personaje está en estado *"Conversando"* pasa a estado *"Esperando"* y viceversa.

A continuación mostramos el método ejecutarAccion dividido por las distintas acciones que se pueden realizar sobre el personaje editado.

```
public Acciones ejecutarAccion(String idA, String idO, String idPj) {
try {
  if ("HABLAR".equals(idA)) {
     if (estado.equals("CONVERSANDO")) {
         return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                  Habitacion h = conseguirHabitacion(idHab);
                  mostrarResultado(h, generaNotificacion(idPJ,"Esta hablando con otro
                              alumno"));
            }

            public boolean interpretaCondicion(Habitacion hab1,
                                 Habitacion hab2, String idPJ) {
                  return true;
            }
         };
     }
     else {
         Acciones a = teRespondo(actual.getNombre(), idPj);
          ObjetosRefresco re = a.generaRefresco(h, idPj);
         a.mostrarResultado(h, re);
         return a;
     }
  }
```

En el caso de la acción Hablar primero comprobamos si el personaje está en estado *"Conversando"* o no. Si está conversando generamos la notificación *"Está hablando con otro alumno"* y la mostramos, si no está conversando creamos la acción a y ejecutamos el método teRespondo que explicaremos más adelante.
Acto seguido generamos un objeto de tipo ObjetoRefresco y mostramos el resultado, esto se tuvo que hacer debido a que encontramos problemas con acciones realizadas durante conversaciones con los personajes ya que los objetos involucrados no se actualizaban hasta que se desconectaba y volvía a conectar el jugador, por ejemplo al meter un objeto en el inventario como resultado de una conversación éste se metía pero no aparecía en la interfaz hasta que no se reconectaba el jugador. Explicamos este problema mejor en el apartado *4.6. Problemas encontrados durante el desarrollo*.

```
else if ("MIRAR".equals(idA)) {
    return new Acciones() {
         public void ejecutar(String idHab, String idPJ) {
             Habitacion h = conseguirHabitacion(idHab);
             NotificacionTextual n = new NotificacionTextual(mirar, idPJ);
             ObjetoActualizacion oA = new ObjetoActualizacion();
             LinkedList lista = new LinkedList();
             lista.add(n);
             oA.setNotificacionesTextuales(lista);
             mostrarResultado(h, oA);
         }

         public boolean interpretaCondicion(Habitacion hab1,
                         Habitacion hab2, String idPJ) {
             return true;
         }
    };
```

La acción Mirar es trivial, generamos una notificación textual con el texto contenido en la variable de tipo String mirar y creamos un un objetoActualización con dicha notificación para mostrar el resultado posteriormente.

La acción usar la vamos a mostrar en cuatro fragmentos de código: búsqueda del objeto utilizado, comprobación de si es el objeto1, comprobación de si es el objeto2 y generar una notificación en caso contrario.

```
else if ("USAR".equals(idA)) {
    Habitacion h = Acciones.conseguirHabitacion(idHab);
    String nom;
    ObjetoNoPersonajeClase oNPC = null;
    if (idO != null) {
        oNPC = h.buscarObjeto(idO, idPj);
        if (oNPC != null) {
             String tipo = oNPC.getTipo();
             int ind = tipo.lastIndexOf(".");
             nom = tipo.substring(ind + 1);
        }
         else
             nom = "Nada";
    }
    else
         nom = "Nada";
    Acciones a = null;
```

Lo primero que hacemos al identificar la acción Usar es buscar el objeto que hemos usado sobre el personaje, para ello obtenemos la habitación en la que estamos y ejecutamos el método *buscarObjeto()* sobre dicha habitación. Este métodos nos devolverá el objeto si lo ha encontrado (ya sea en la habitación o en el inventario del jugador) y null en caso contrario.
Si el objeto es distinto de null, es decir lo ha encontrado, cogemos el tipo del objeto y tomamos la subcadena situada desde el último punto hasta el final del string, que corresponderá con el nombre del objeto. Si no ha encotrado el objeto ponemos de nombre “Nada” puesto que es posible que hubiese algún posible error si usasemos la cadena vacía en caso de omisión de datos en el editor.
Por último creamos una nueva acción a sobre la que ejecutar el método *teRespondo()* en caso de ser necesario.

A continuación comprobamos si el objeto usado coincide con el primer objeto que se puede usar con el personaje y si dicho objeto está en el estado en el que se puede usar, además hay que comprobar que el personaje editado no esté hablando con otro jugador en ese momento:

```
if ((nom.equals(ob1)) && (oNPC.getEstado().equals(estado1)) &&
        (estado.equals("ESPERANDO")))
{
   usado1 = true;
   a = teRespondo(conv1.getNombre(), idPj);
   usado1 = false;
}
```

Si se cumplen las condiciones arriba expuestas ponemos usado1 a true ya que hemos usado el primer objeto e igualamos la acción a al método *teRespondo()* al cual pasamos por parámetro el nombre de la conversación que se lanza al usar el primer objeto y el id del jugador que lo usó.

La comprobación de si el objeto usado pertenece con el segundo posible objeto a usar es idéntica a la comprobación del primer objeto, con la salvedad claro de que en este caso se usan las variables relacionadas con el segundo objeto: ob2 para el nombre de objeto, estado2 para el estado en que se puede usar, usado2 para indicar que se ha usado el objeto y conv2 para obtener el nombre de la conversación que se lanzará:

```
else if ((nom.equals(ob2)) && (oNPC.getEstado().equals(estado2)) &&
       (estado.equals("ESPERANDO")))
{
    usado2 = true;
    a = teRespondo(conv2.getNombre(), idPj);
    usado2 = false;
}
```

Por último si no se ha usado ninguno de los objetos posibles generamos la notificación *"No puedo hacer eso"* y mostramos el resultado.

```
else
   return new Acciones() {
      public void ejecutar(String idHab, String idPJ) {
          Habitacion h = conseguirHabitacion(idHab);
          mostrarResultado(h, generaNotificacion(idPJ,
                       "No puedo hacer eso"));
      }

      public boolean interpretaCondicion(Habitacion hab1,
                        Habitacion hab2, String idPJ) {
           return true;
      }
   };
return a;
}
```

Terminamos el código de *ejecutarAccion()* indicando que si la acción realizada no es ni hablar, ni mirar ni usar se devuelva la notificación *"No puedo hacer eso"*:

```
else
{
   return new Acciones() {
      public void ejecutar(String idHab, String idPJ) {
          Habitacion h = conseguirHabitacion(idHab);
          mostrarResultado(h, generaNotificacion(idPJ, "No puedo hacer eso"));
      }

      public boolean interpretaCondicion(Habitacion hab1,
                       Habitacion hab2, String idPJ) {
          return true;
      }
   };
}
}
catch (Exception e)
  {
        System.err.println("Excepción en ejecutarAccion");
        e.printStackTrace();
        return null;
  }
}
```

Por último la clase PersonajeEditor contiene el método *teRespondo()* el cual como indicabamos anteriormente se lanza tanto al hablar con el personaje como al usar uno de los objetos posibles sobre el personaje.

```
public Acciones teRespondo(String opcion, String idPJ) {
    if (estado.equals("ESPERANDO")) {
         estado = "CONVERSANDO";
    }
    for (int i = 0; i < convs.size(); i++){
        actual = (TransferConversacion) convs.elementAt(i);
        if (actual != null){
        if (opcion.equals(actual.getNombre())){
                if (actual.isTerminaConv())
                        estado = "ESPERANDO";
                return new Acciones() {
                 public void ejecutar(String idHab, String idPJ) {
                             Habitacion h = conseguirHabitacion(idHab);
                             ObjetoComunicacion o = new ObjetoComunicacion();
                             LinkedList conversacion = new LinkedList();
```

En primer lugar cambiamos el estado del personaje editado a *"Conversando"*. Una vez hecho esto recorremos el vector de conversaciones y vamos tomando una a una, si la conversación actual es distinta de null comprobamos si la opción que se ejecutó en *teRespondo()* coincide con el nombre de la conversación actual en caso de ser así quiere decir que tenemos que reproducir esta conversación.
Lo primero que hacemos una vez tenemos la conversación que hay que mostrar es comprobar si es conversación final o no, si es conversación final cambiamos el estado del personaje a *"Esperando"* para que otros jugadores puedan hablar con él. Posteriormente devolvemos una nueva acción en la que lo primero que hacemos es conseguir la habitación en la que está el personaje, crear un objeto de tipo ObjetoComunicación para poder mostrar la conversación por la interfaz gráfica y crear un objeto de tipo LinkedList en el que iremos añadiendo la conversación línea a línea.

```
o.setEmpiezaJugador(actual.isEmpiezaJug());
int lineaJug = 0; int lineaPj = 0;
for (int i = 0; i < actual.getTextoJug().size() + actual.getTextoPj().size();   i++)
{
        if (actual.isEmpiezaJug() && i%2 == 0)
                conversacion.add(actual.getTextoJug().elementAt(lineaJug++));
        else if (actual.isEmpiezaJug() && i%2 != 0)
                conversacion.add(actual.getTextoPj().elementAt(lineaPj++));
        else if (!actual.isEmpiezaJug() && i%2 == 0)
                conversacion.add(actual.getTextoPj().elementAt(lineaPj++));
        else
                conversacion.add(actual.getTextoJug().elementAt(lineaJug++));
}
o.setConversacionFija(conversacion);
```

Lo siguiente que hacemos es indicar en el objetoComunicación quien empezará hablando, esto lo obtenemos de la conversación actual. Creamos dos variables auxiliares que nos servirán para añadir las líneas de la conversación de manera correcta en el LinkedLista conversación.
Recorremos un for de tamaño igual a la suma de las líneas de conversación del jugador y del personaje (que será en definitiva el tamaño de la conversación). Si empieza hablando el jugador le corresponderán las líneas pares y si no las impares, por lo tanto comprobamos si empieza hablando y si el índice es par y así podemos generar la conversación con el orden correcto.
Por último fijamos la conversación en el objetoComunicación.

Una vez tenemos la conversación fijada correctamente pasamos a comprobar si hay que añadir posibles opciones de contestación, para ello obtenemos el vector de opciones de la conversación actual y comprobamos si su tamaño es mayor que cero:

```
if (actual.getOpciones().size() > 0){
   LinkedList opciones = new LinkedList();
   for (int i = 0; i < actual.getOpciones().size(); i++){
        String opcion = actual.getOpciones().elementAt(i).toString();
        if (opcion != null)
        {
           opciones.add(new Frase(opcion, opcion));
        }
        o.setOpciones(opciones);
   }
   o.setIdPJ(idPJ);
   o.setNombrePNJ(nombre);
   mostrarResultado(h, o);
}
```

Si la conversación tiene opciones de respuesta creamos un LinkedList opciones y recorremos con un for el vector de opciones de la conversación, para cada opción distinta de null que encontremos añadimos una nueva Frase en en LinkedList opciones. Las frases añadidas constan de dos valores, en el primero está el texto de la respuesta en sí y en el segundo el id asignado a dicha respuesta, en nuestro caso hemos decidido usar como índice de la respuesta el propio texto de la misma, para facilitar la petición de datos y el entendimiento de las conversaciones desde el editor.
Una vez tenemos la conversación y las opciones en el objetoComunicación indicamos el id del jugador y el nombre del personaje que tiene la conversación y mostramos el resultado.

Tras completar todo lo relacionado con la conversación nos queda tratar las posibles acciones asociadas a la conversación o bien al uso del objeto que lanzó la conversación. Estas posibles acciones se dividen en tres grupos, las producidas por la propia conversación (actual), las que se producen al usar el primer objeto sobre el personaje (act1) y las que se producen al usar el segundo objeto sobre el personaje (act2):

```
if (actual.getAccion().compareTo("Cambiar el estado del objeto") == 0)
        cambiarEstadoObjeto(h, actual.getDest() + idPJ, idPJ, actual.getDest()
                + " ha cambiado de estado");
if (actual.getAccion().compareTo("Meter objeto en el inventario") == 0
        && objetos !=  null)
{
        TransferObjeto cogido
        =(TransferObjeto)objetos.elementAt(Integer.parseInt(actual.getDest()));
        cogido.setPosX("0"); cogido.setPosY("0");
        ObjetoEditor obCogido = new ObjetoEditor(cogido, null, null);
        meterEnInventario(h, idPJ,
           "agm.objetos.objetosNoPersonaje.ObjetoEditor",
           obCogido, “Has conseguido un " +  obCogido.getNombre());
}
if (actual.getAccion().compareTo("Sacar objeto del inventario") == 0)
        sacarDeInventario(h, actual.getDest()+idPJ, idPJ, "Ya no tengo " +
        actual.getDest());
if (actual.getAccion().compareTo("Cambiar personaje de habitacion") == 0)
{
         Habitacion hDest = conseguirHabitacion("H" + actual.getDest());
         cambioHabitacion(h, hDest, idPJ);
         ObjetosRefresco oR = generaRefresco(hDest, idPJ);
         mostrarResultado(h, oR);
}
```

Como se aprecia en el código hacemos comparaciones con la acción de actual y si coincide con alguna se hace dicha acción. Estas acciones y comprobaciones son prácticamente iguales a las que se hacían en el apartado *4.4.1. ObjetoEditor* para realizar las acciones de los objetos editados, con la salvedad de que en este caso las variables utilizado para obtener el objeto cogido, la habitación de destino y demás se obtienen de los atributos de la conversación.

En el caso de las acciones debidas al uso del primer objeto sobre el personaje, el código es exactamente igual salvo, por supuesto, las variables relacionadas con el elemento involucrado en la acción y los mensajes a mostrar, que en este caso se obtienen directamente de act1 y dest1:

```
if (usado1){
  if (act1.compareTo("Cambiar el estado del objeto") == 0)
     cambiarEstadoObjeto(h, dest1 + idPJ, idPJ, dest1 + " ha cambiado de estado");
  if (act1.compareTo("Meter objeto en el inventario") == 0 && objetos != null)
  {
        transferObjeto cogido =
               (TransferObjeto)objetos.elementAt(Integer.parseInt(dest1));
        cogido.setPosX("0"); cogido.setPosY("0");
        ObjetoEditor obCogido = new ObjetoEditor(cogido, null, null);
        meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.ObjetoEditor",
               obCogido, "Has conseguido un " + obCogido.getNombre());
  }
  if (act1.compareTo("Sacar objeto del inventario") == 0)
        sacarDeInventario(h, dest1+idPJ, idPJ, "Ya no tengo " + dest1);
  if (act1.compareTo("Cambiar personaje de habitacion") == 0)
  {
       Habitacion hDest = conseguirHabitacion("H" + dest1);
       cambioHabitacion(h, hDest, idPJ);
        ObjetosRefresco oR = generaRefresco(hDest, idPJ);
        mostrarResultado(h, oR);
   }
}
```

Para las acciones debidas al uso del segundo objeto repetimos nuevamente el código usando ahora las variables act2 y dest2 para obtener los datos necesarios para realizar las acciones sobre el objeto involucrado:

```
 if (usado2){
       if (act2.compareTo("Cambiar el estado del objeto") == 0)
        cambiarEstadoObjeto(h, dest2 + idPJ, idPJ, dest2 + " ha cambiado de estado")
        if (act2.compareTo("Meter objeto en el inventario") == 0 && objetos != null)
        {
           TransferObjeto cogido =
                (TransferObjeto)objetos.elementAt(Integer.parseInt(dest2));
           cogido.setPosX("0"); cogido.setPosY("0");
           ObjetoEditor obCogido = new ObjetoEditor(cogido, null, null);
           meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.ObjetoEditor",
                       obCogido, "Has conseguido un " + obCogido.getNombre());
        }
        if (act2.compareTo("Sacar objeto del inventario") == 0)
                sacarDeInventario(h, dest2+idPJ, idPJ, "Ya no tengo " + dest2);
        if (act2.compareTo("Cambiar personaje de habitacion") == 0)
        {
                Habitacion hDest = conseguirHabitacion("H" + dest2);
                cambioHabitacion(h, hDest, idPJ);
                ObjetosRefresco oR = generaRefresco(hDest, idPJ);
                mostrarResultado(h, oR);
        }
}
```

4.5. Creando una misión paso a paso

En esta sección se describe una historia y el proceso seguido en el Editor de misiones para la realización de la misma.

4.5.1. Guión de la historia

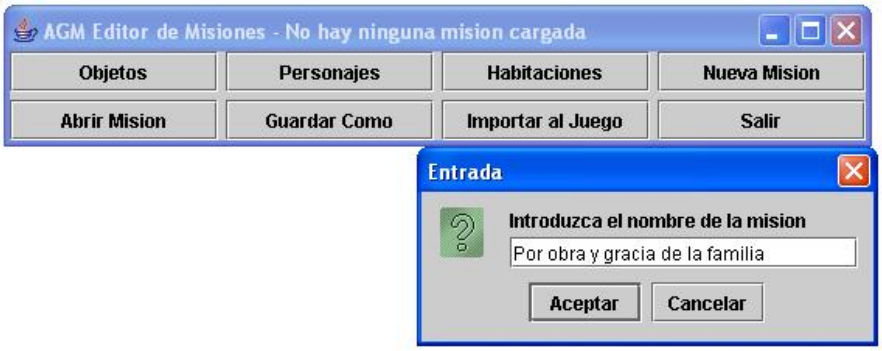
La historia se llama *Por obra y gracia de la familia.* En esta historia el jugador deberá sobornar al profesor de la asignatura DGT para que le apruebe pese al bochornoso examen que realizó. Para ello le vendrá muy bien la ayuda de su padre, el cual es agente de movilidad, y que de modo poco honesto y tras una pequeña conversación accederá a darle una multa falsa a su hijo de poca cuantía pensando que es para una pequeña broma.
El jugador con la multa en su poder tendrá que conseguir un bolígrafo del mismo color que los números de la multa para trucarla, el cual conseguirá cogiéndolo de una caja de bolígrafos que hay en la biblioteca y añadir unos cuantos ceros a la cantidad a pagar. Con la multa en posesión el jugador irá al despacho del profesor de DGT y tras mostrarle la multa sobornará al profesor para conseguir el aprobado.

En resumen los elementos necesarios para crear la misión serán:

- Cuatro habitaciones: Exterior, Pasillo, Entrada de la Biblioteca y Biblioteca.
- Cuatro objetos: Multa, Boligrafo, Caja de Bolígrafos y AprobadoDGT
- Dos personajes no jugadores: Agente de Movilidad, ProfesorDGT

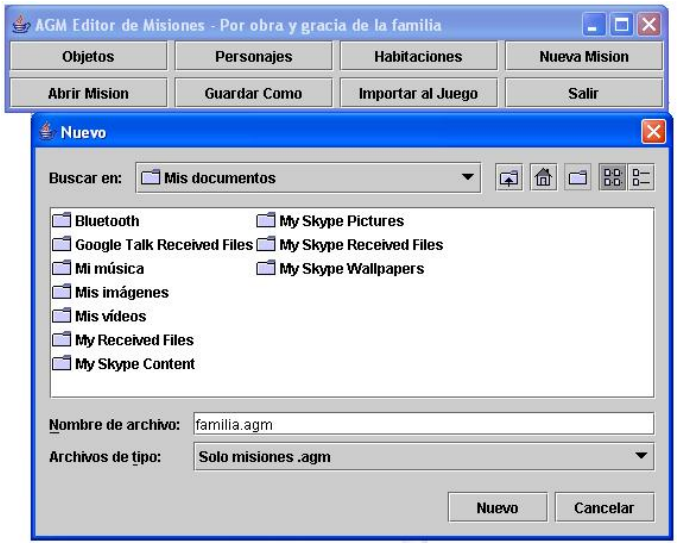
4.5.2. Crear una misión nueva

El primer paso para crear la misión será ejecutar el Editor de misiones y dar un nombre a la misión de tal forma que tengamos un fichero .agm sobre el que trabajar, como se aprecia en la imagen 4.22.



4.22 Dando nombre a la misión

Una vez elegido el nombre se nos pedirá donde queremos guardar la misión como se muestra en la imagen 4.23:



4.23 Guardar una misión

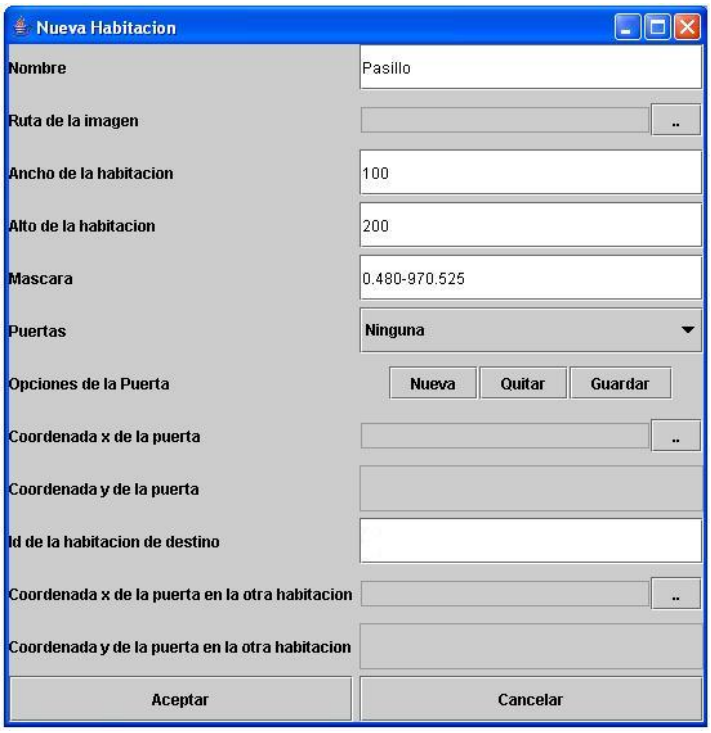
4.5.3. Creando las habitaciones

Una vez tenemos la misión creada vamos a empezar generando las habitaciones que necesitaremos para la misión que como comentabamos en el apartado 4.5.1 Guión de la historia, serán cuatro.
De las habitaciones que vienen ya creadas en la aplicación original usaremos el Hall desde el cual se accederá al pasillo y la entrada de la biblioteca, y vamos a añadir el exterior de la facultad donde estará el agente de movilidad padre del alumno, un pasillo desde el que se accederá al despacho del profesor, la entrada a la biblioteca y la biblioteca en sí donde encontraremos la caja de bolígrafos con el bolígrafo necesario.

Al clickear sobre habitaciones en el menú principal aparecerá la gui de las habitaciones, que se muestras en la imagen 4.24, los botones << y >> que se aprecian en la imagen sirven para pasar a la habitación anterior o posterior respectivamente:

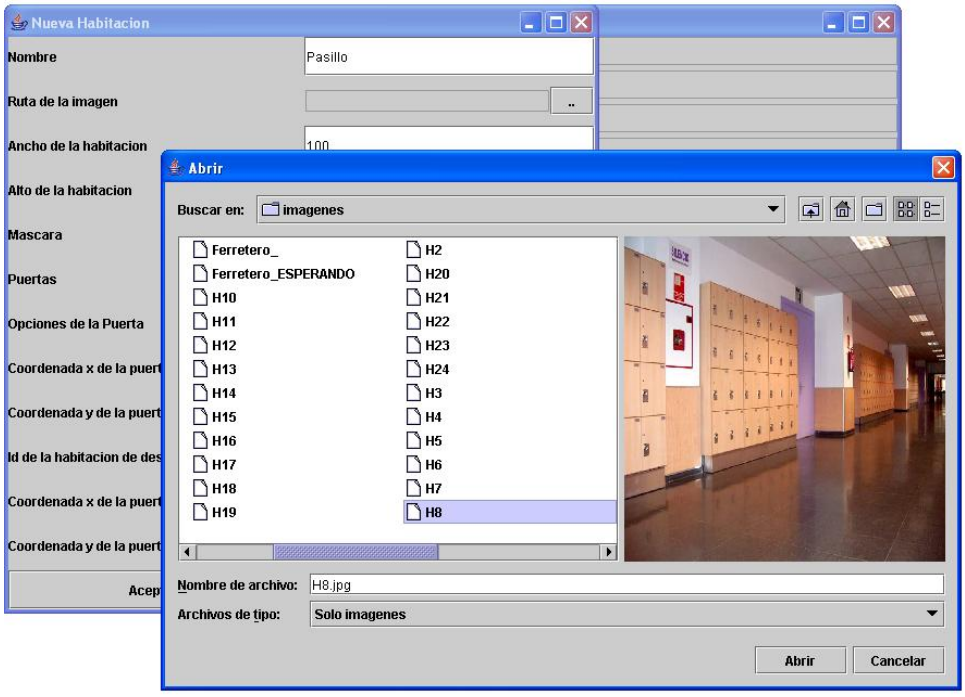
4.24 Interfaz gráfica de las habitaciones

Como no hemos creado habitaciones aún todos los campos están vacíos, pulsamos en Nueva Habitación (imagen 4.25) para crear la primera habitación, como ejemplo en el tutorial crearemos el pasillo puesto que tendrá tres puertas y servirá para mostrar como se controlan en el editor.



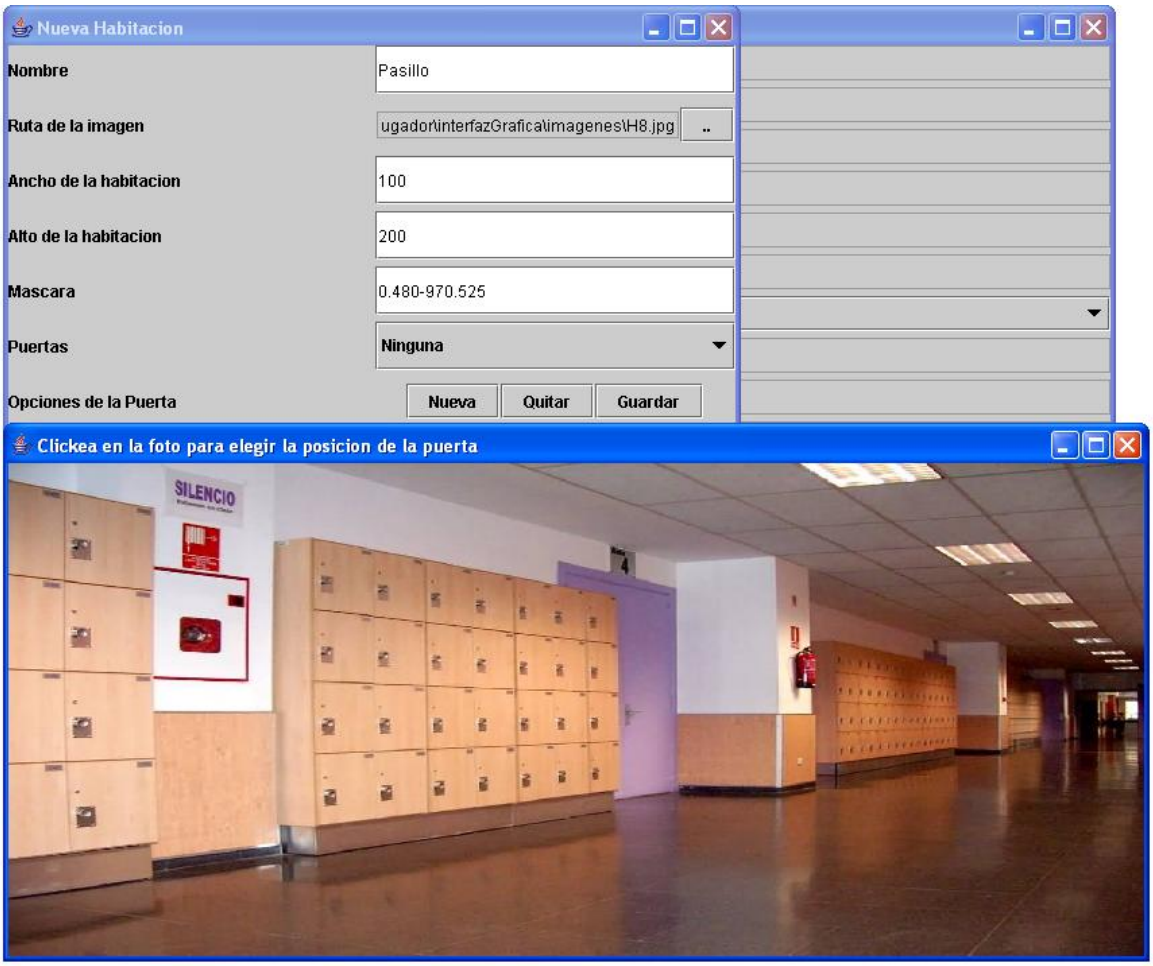
4.25 Creando una habitación nueva

Una vez tenemos los datos generales de la habitación pasamos a elegir la imagen que se mostrará en la habitación pulsando sobre el botón **..** de Ruta de la imagen, como se muestras en la imagen 4.26.



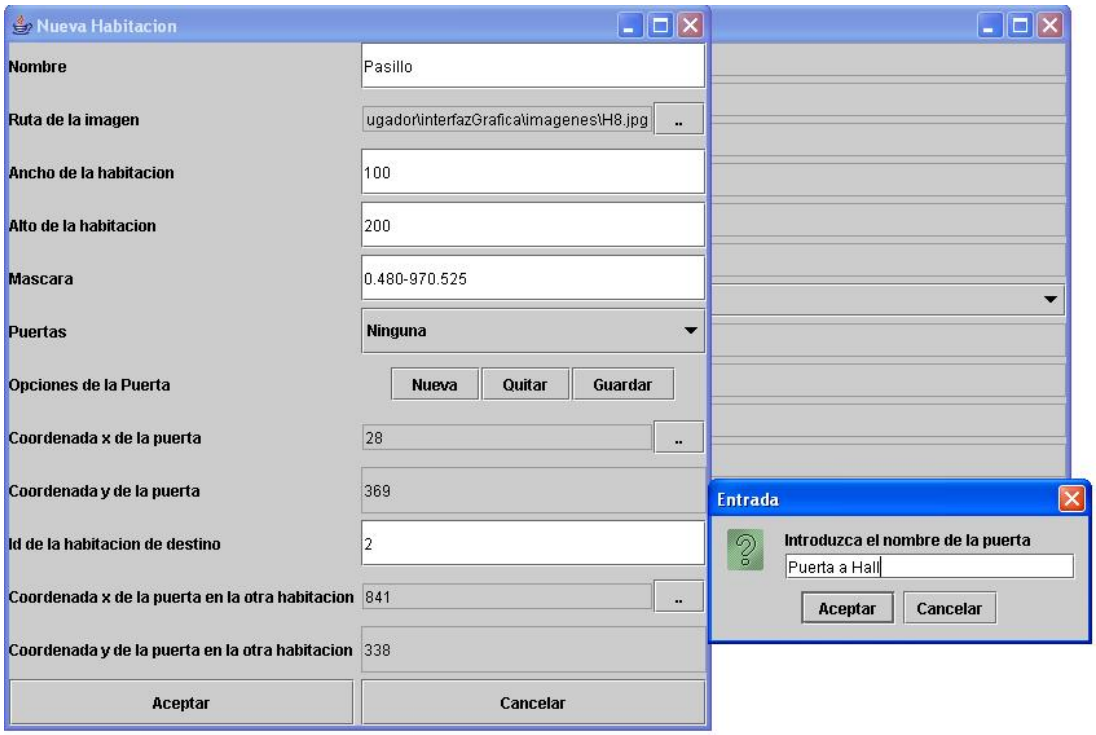
4.26 Eligiendo la imagen de la habitacion

Con todos los datos necesarios para crear la habitación escritos, pasamos a crear las puertas para ello introduciremos los datos de los campos de texto inferiores relativos a puerta y pulsaremos sobre Nueva una vez estén escritos.
Como se puede apreciar se necesita indicar las coordenadas de la puerta en la habitación actual y en la de destino, mostraremos en este caso el proceso pero lo obviaremos en el resto de puertas. Al pulsar el botón **..** de Coordenadas x de la puerta aparecerá la gui de coordenadas puerta, imagen 4.27, para que clickeemos en el lugar donde queremos situar la puerta.



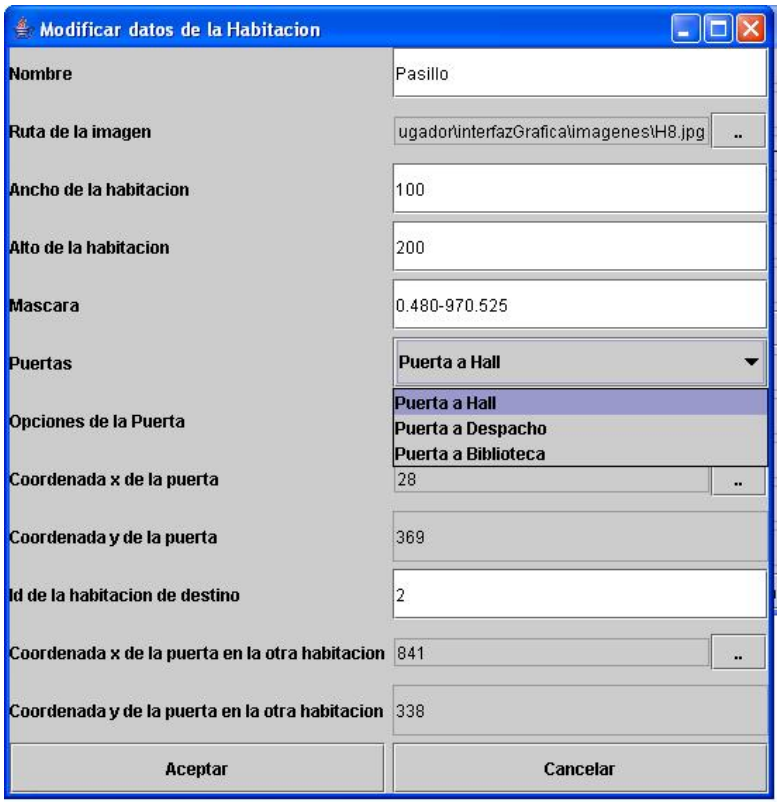
4.27 Coordenadas de la puerta

Al pinchar en el punto deseado se rellenarán automáticamente los campos de texto de las coordenadas de la puerta en la habitación actual, como se explicaba en el apartado *4.3.5 Captura de coordenadas de un personaje* pero con las puertas, repetiremos el mismo proceso para la habitación de destino para la cual se cargará la imagen H2 que corresponde con el hall, una vez tenemos todos los datos de la puerta pulsamos en Nueva y se nos pedirá que introduzcamos el nombre de la puerta creada, como muestra la imagen 4.28.



4.28 Dando nombre a la puerta

Con esto quedará creada la puerta y se añadirá al ComboBox Puertas, imagen 4.29, por si posteriormente la queremos eliminar con el botón Quitar o modificar alguno de sus datos cambiando los campos de texto relacionados y pulsando sobre Guardar.



4.29 ComboBox de las puertas

Al pulsar sobre cualquiera de las puertas del ComboBox automáticamente se actualizarán los campos de texto correspondientes con los valores de esa puerta. Cabe destacar que para crear las puertas necesitamos conocer el ID de la habitación de destino por lo tanto lo normal es crear las habitaciones primero y después añadir las puertas sabiendo los Ids de cada una de las habitaciones, pero en este tutorial se hace directamente para facilitar el entendimiento del mismo.
Una vez tenemos todos los datos del Pasillo pulsamos en Aceptar y volveremos a la interfaz gráfica general de las habitaciones pero veremos como ahora se muestran los datos de la habitación que acabamos de crear, imagen 4.30.

4.30 Datos de la habitación recien creada

Al igual que en Nueva Habitación aquí también se mostrarán los datos de cada una de las puertas al seleccionarlas en el ComboBox. Como se puede apreciar en la imagen 4.30, se ha asignado el id 24 a la primera habitación que hemos creado, esto se hace así porque en la aplicación original AGM ya existen 23 habitaciones y como se comentaba en la introducción no se eliminan al importar la misión si no que se acoplan.

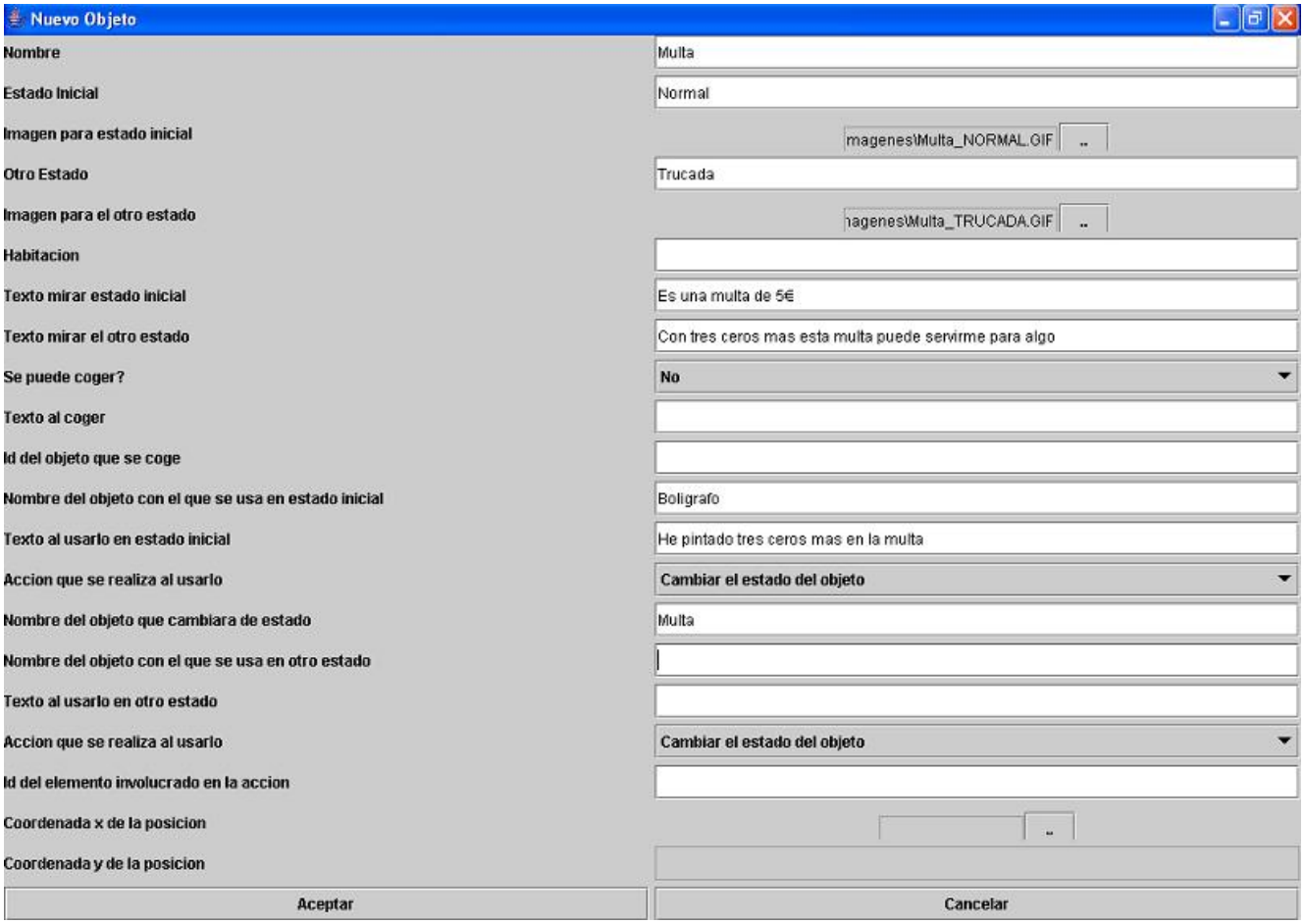
El resto de habitaciones de la misión, es decir, el exterior de la facultad, el despacho del profesor, la entrada a la biblioteca y la biblioteca en sí se crearán de la misma forma, pero obviamos el poner el proceso en el tutorial puesto que es trivial tras haber visto como se crea la primera habitación.

4.5.4. Creando los objetos

Cuando ya tenemos creadas todas las habitaciones necesarias en la historia pasamos a crear los objetos. En este caso serán cuatro objetos: multa, bolígrafo, caja de bolígrafos y aprobadoDGT.
En el tutorial solo mostraremos el proceso para crear la multa y la caja de bolígrafos puesto que tienen las acciones más significativas.
Pulsamos el botón objetos de la interfaz principal y aparecerá la interfaz principal (imagen 4.31) de los objetos, al igual que con las habitaciones los campos estarán vacíos al no haber objetos aún.

4.31 Interfaz principal de los objetos

Pulsamos sobre NuevoObjeto para crear el objeto nuevo, mostramos la siguiente imagen 4.32 con los datos ya introducidos para acortar.

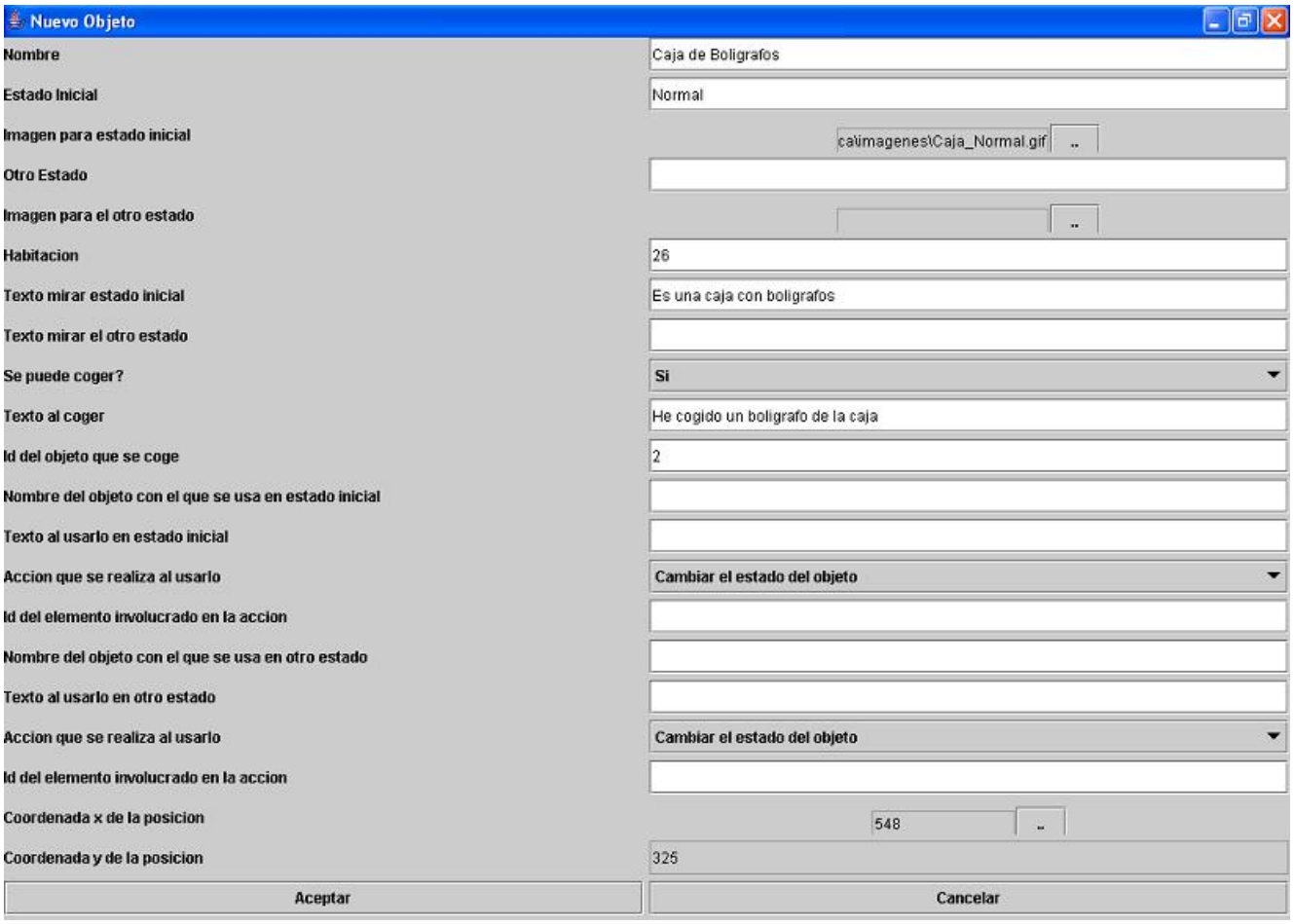


4.32 Datos de la Multa

Como se puede apreciar en la imagen 4.32 no indicamos habitación puesto que nos la dará un personaje no jugador, en este caso el padre del alumno, ponemos que no se puede coger puesto que esta acción se refiere a coger el objeto directamente y en este caso nos lo dan.
Ponemos el nombre del objeto con el que se usa en estado inicial que es Boligrafo que aunque aun no esté creado le pondremos ese nombre posteriormente.
El ComboBox en el que se muestra “Cambiar el estado del objeto” es el que nos indica que acción se realizará al usarlo, que como se citaba anteriormente nos permite cuatro posibles acciones: cambiar el estado del objeto, meter objeto en inventario, sacar objeto del inventario y cambiar al jugador de habitación.
Por último aunque la multa trucada, es decir en estado secundario, se usará con el profesor no lo indicamos aquí puesto que estas acciones son entre objeto y objeto, el uso de la multa sobre el profesor se indicará al crear el profesor.

Con esto tenemos creado el objeto Multa, a continuación vamos a mostrar como se crea el objeto caja de bolígrafos puesto que tiene algunas opciones distintas que merece la pena reseñar.

Al terminar con la Multa pulsamos en Aceptar y volveremos a la interfaz principal de objetos esta vez mostrando ya los datos de Multa, imagen 4.33, pulsamos de nuevo en Nuevo Objeto para crear la caja de bolígrafos.

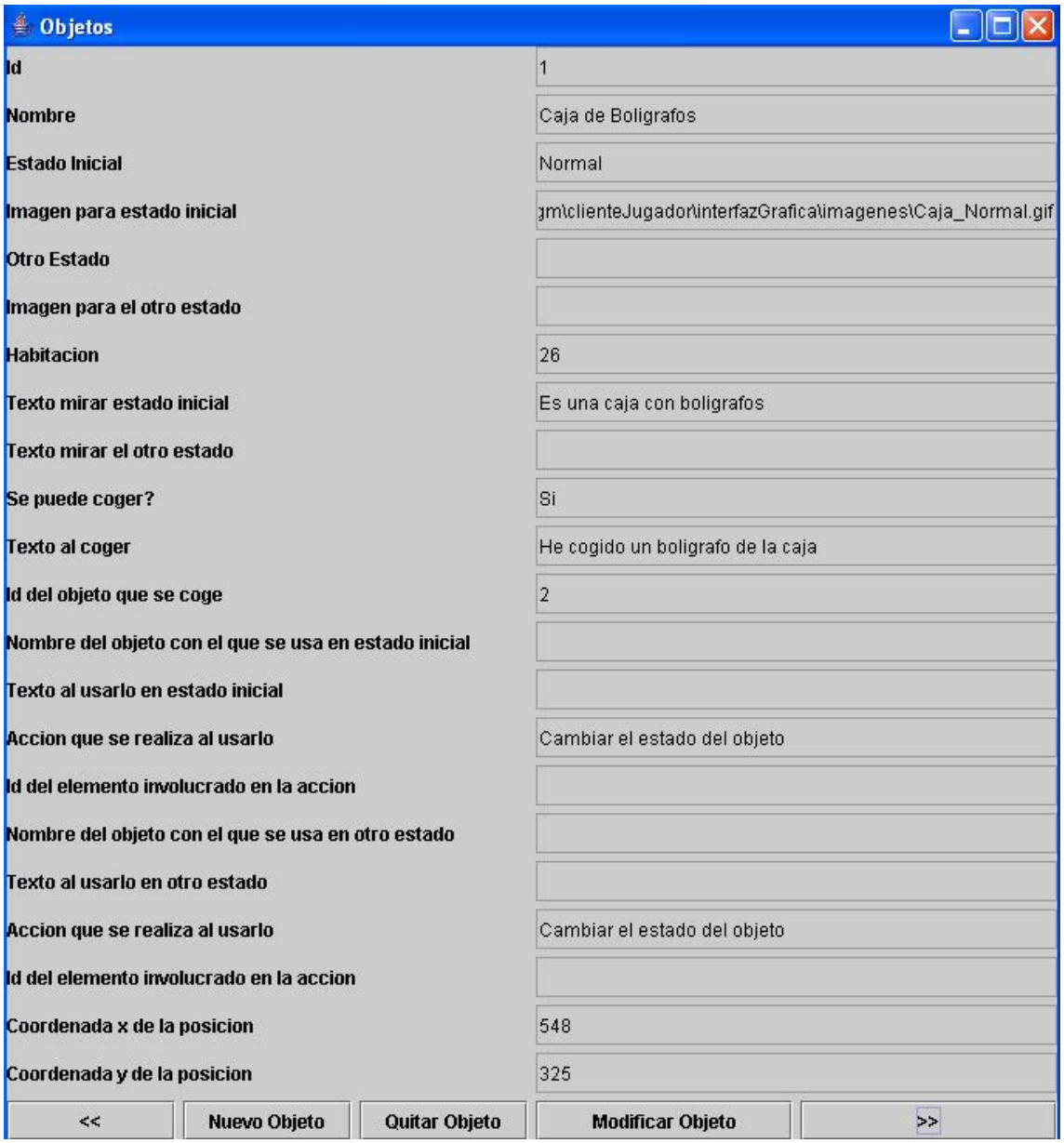


4.33 Datos de la Caja de Bolígrafos

En este caso solo tenemos un estado, sin embargo si que hay que indicar el id de la habitación puesto que es un objeto que está en una habitación, elegimos el id 26 que es el de la biblioteca.
Ahora sí ponemos que se puede coger, por lo tanto escribimos también el texto que se recibe al coger el objeto y el objeto que cogemos, es decir, no tenemos porque coger la caja de boligrafos si no, como por ejemplo en este caso, cogemos un bolígrafo que tiene id 2, esto lo sabemos porque los objetos se empiezan a guardar con id 0 por lo tanto tras crear la multa y la caja de boligrafos el siguiente objeto tendra id 2. Como comentábamos con las habitaciones el proceso normal sería crear primero el bolígrafo y luego la caja de bolígrafos pero no mostraremos como se crea el objeto bolígrafo puesto que es prácticamente igual a como se crea la multa.

Por último, como se aprecia en la imagen 4.33 en este caso si que hay coordenadas de la posición del objeto puesto que está en una habitación, el proceso de captura de las coordenadas es igual que como explicábamos con las habitaciones.

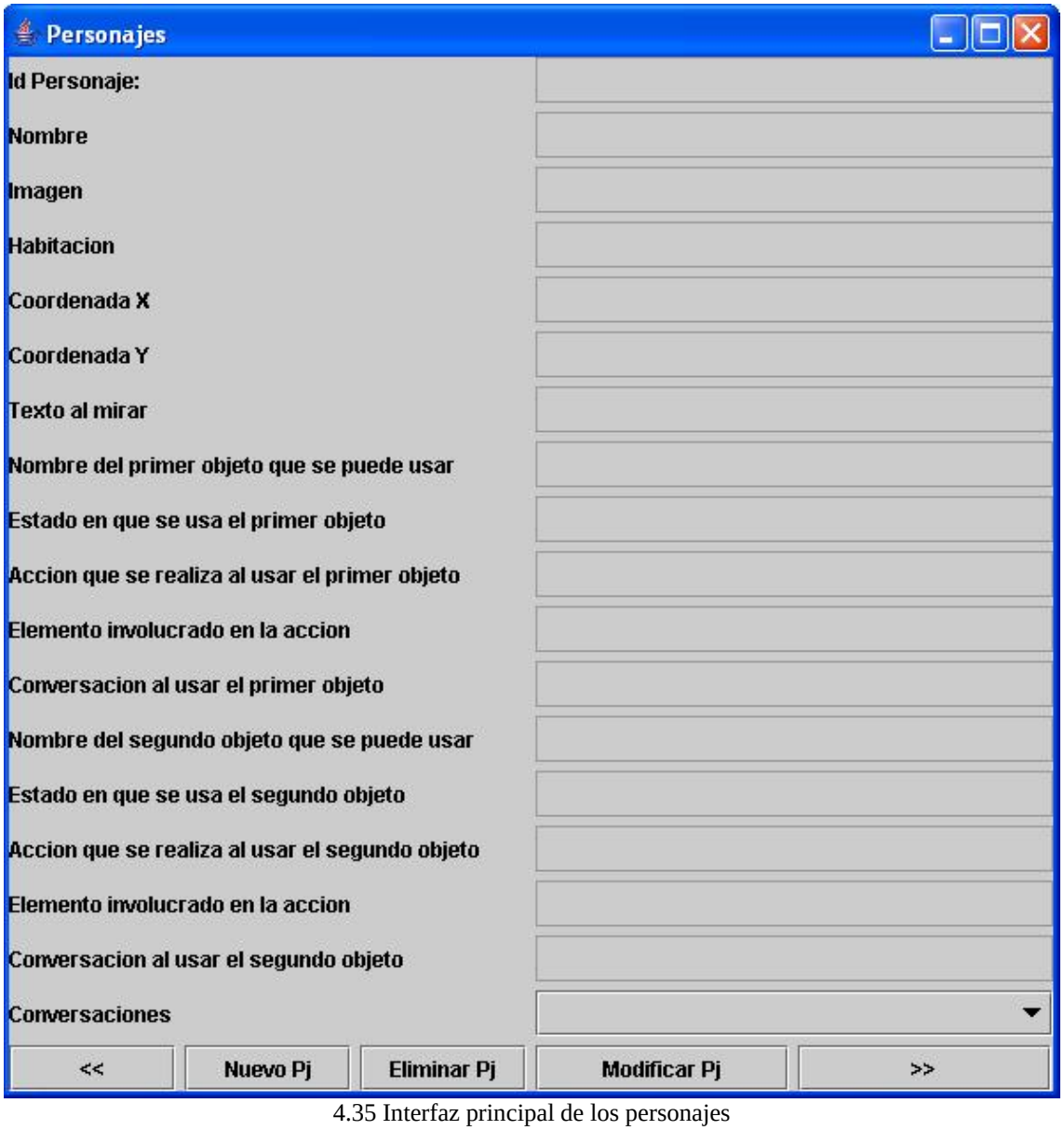
Cuando tenemos la caja de bolígrafos creada pulsamos aceptar y volveremos a la interfaz principal de los objetos en la que se mostrarán los datos de la caja, dado que siempre que se crea un nuevo objeto se muestran automáticamente los datos de éste. Como se aprecia en la imagen 4.34 se ha asignado el id 1 a la caja puesto que hemos creado antes la multa que tiene id 0. Al igual que con la interfaz de las habitaciones los botones << y >> servirán para mostrar el objeto anterior y al posterior al actual. En este caso si pulsasemos << se mostraría la multa y si pulsamos >> saldría un mensaje indicando que no hay más objetos, puesto que aun no está creado el bolígrafo ni el aprobado.



4.34 Datos de la Caja de Bolígrafos recien creada

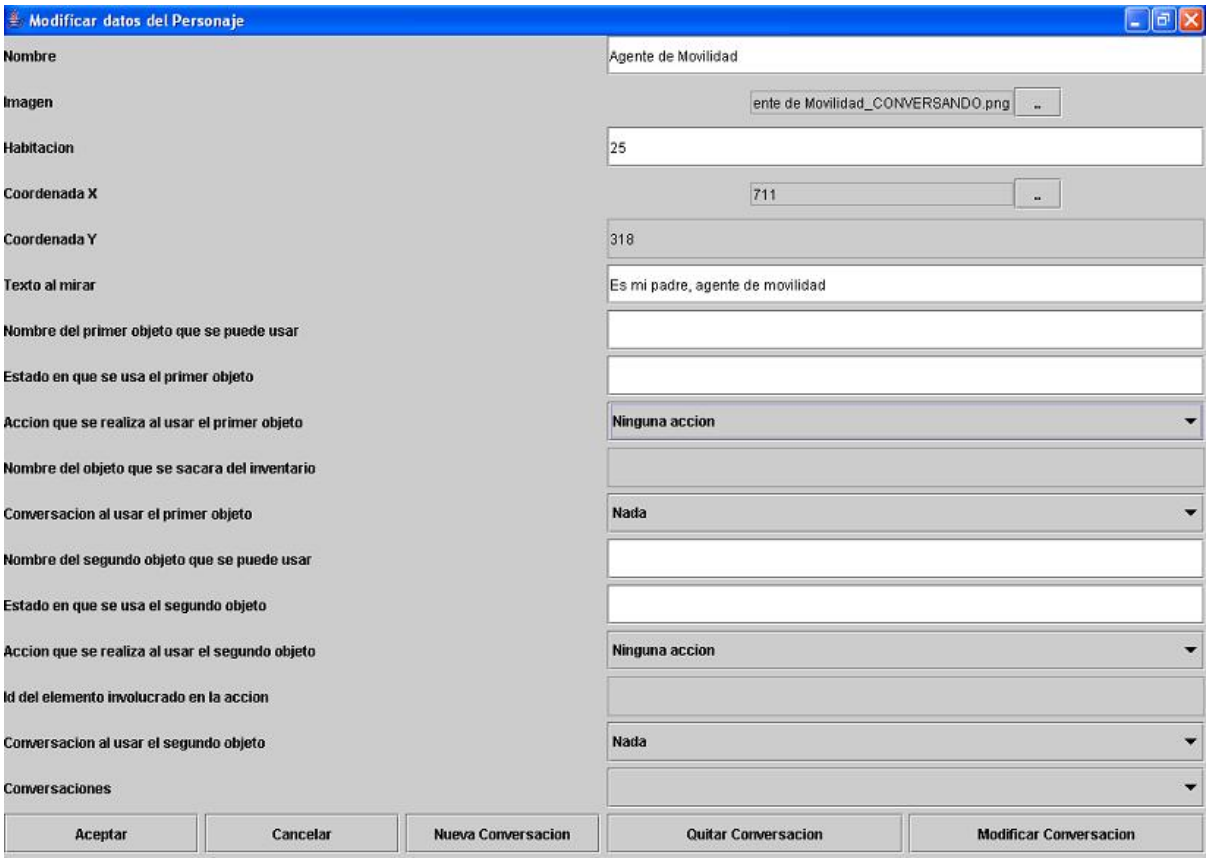
4.5.5. Creando los personajes

Ahora que tenemos creadas las habitaciones y los objetos de la misión solo nos falta crear los personajes. En este caso son dos personajes, el agente de movilidad padre del alumno y el profesor de DGT.
Volvemos a la interfaz principal y clickeamos en Personajes para acceder al menú principal de los personajes, que se muestra en la imagen 4.35.



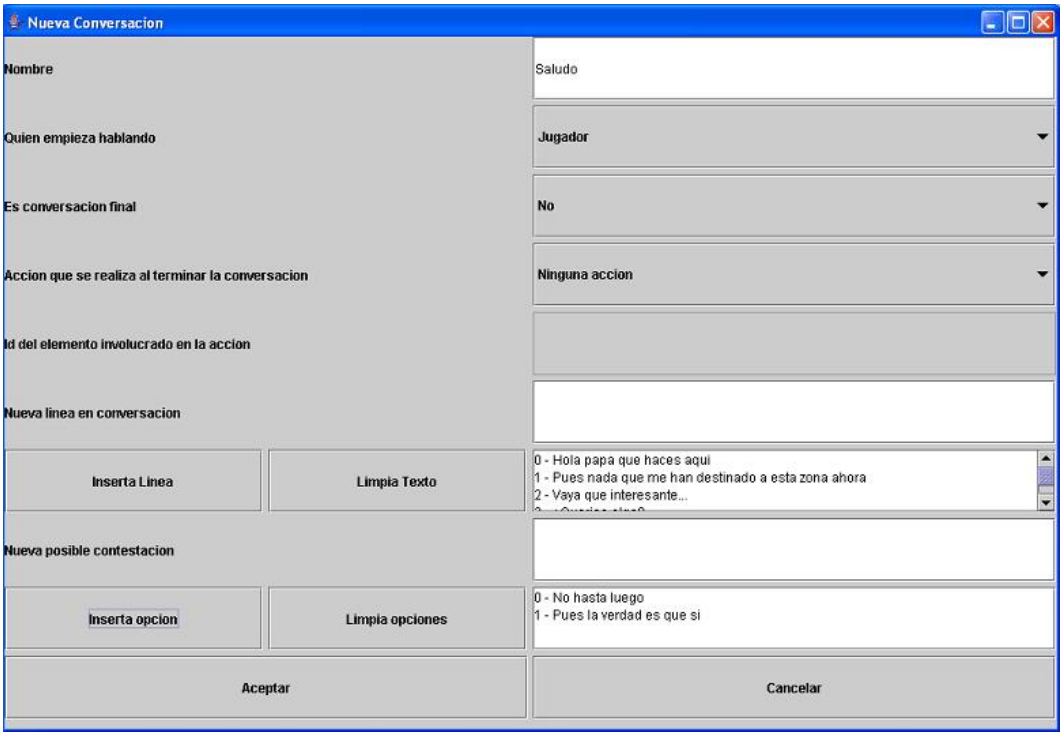
4.35 Interfaz principal de los personajes

Pulsamos en Nuevo Pj y comenzamos a introducir los datos básicos del agente de movilidad, imagen 4.36.



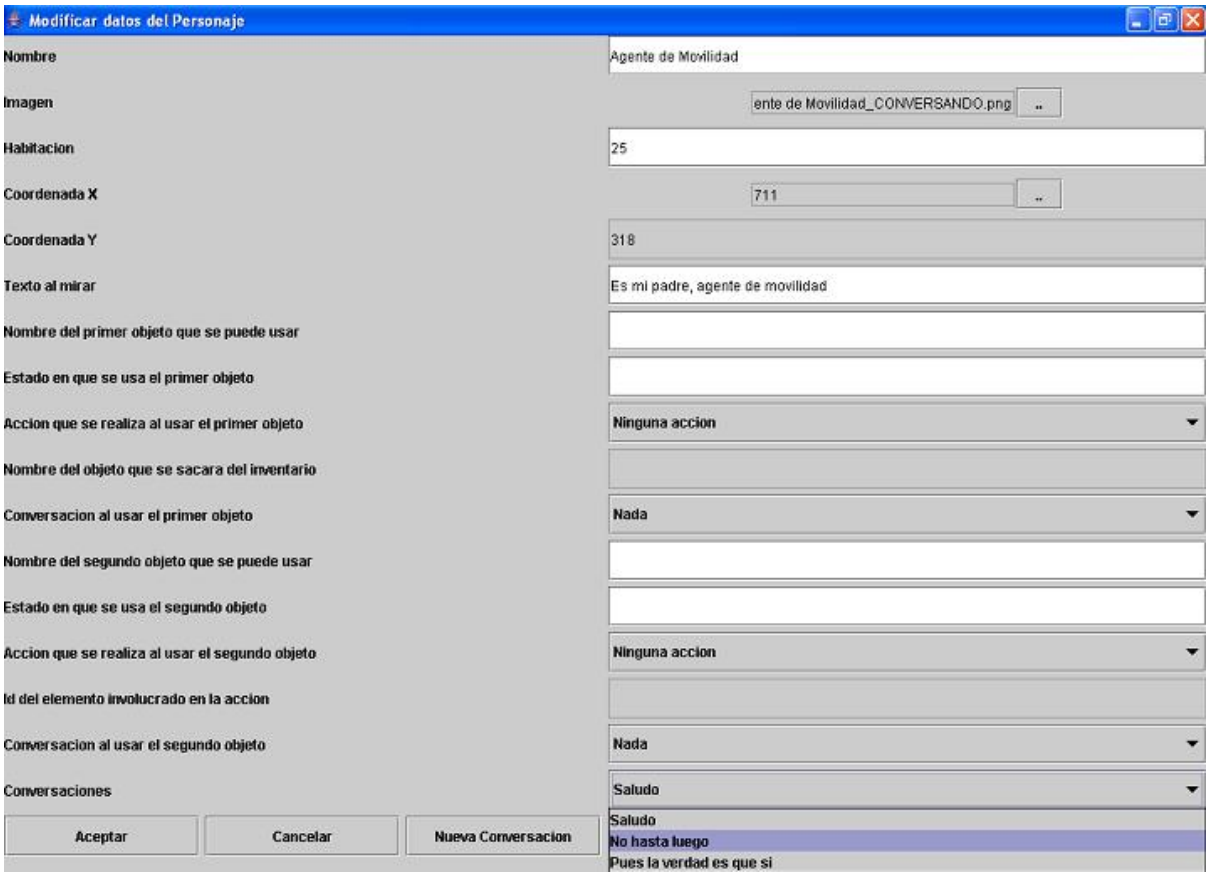
4.36 Datos del Agente de Movilidad

No indicamos el uso de ningún objeto puesto que la multa la recibiremos tras una conversación y cada conversación tiene su propia acción que se puede realizar al final de la misma, por lo tanto pinchamos en Nueva Conversación y pasamos a crear la primera conversación, de nombre Saludo, imagen 4.37.



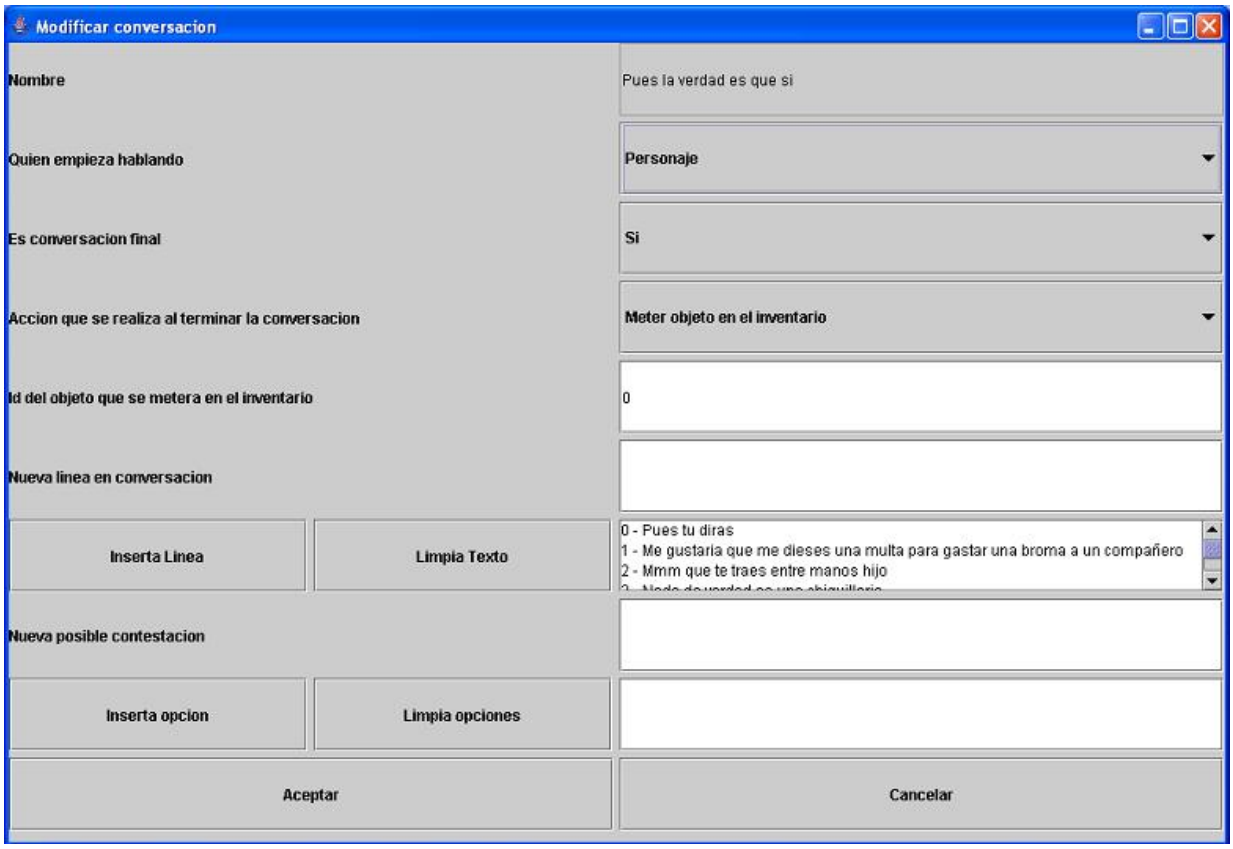
4.37 Datos de la conversación Saludo

Como se aprecia en la imagen 4.37 en el editor de conversaciones disponemos de un campo de texto en el que ir escribiendo cada línea de la conversación, al pulsar en Inserta Linea se añade a la conversación global. De la misma forma añadimos las posibles opciones de respuesta, que se mostrarán en el juego justo después de la conversación.
En este caso tenemos las opciones *No hasta luego* y *Pues la verdad es que si* que serán posibles respuestas a la última línea de la conversación que es *¿Querías algo?*.
Tras pulsar aceptar veremos como automáticamente se han creado aparte de la propia conversación *Saludo* dos conversaciones más que son *No hasta luego* y *Pues la verdad es que si*, imagen 4.38, que serán las que se carguen cuando pulsemos la opción correspondiente en el juego.



4.38 Conversaciones del Agente de Movilidad

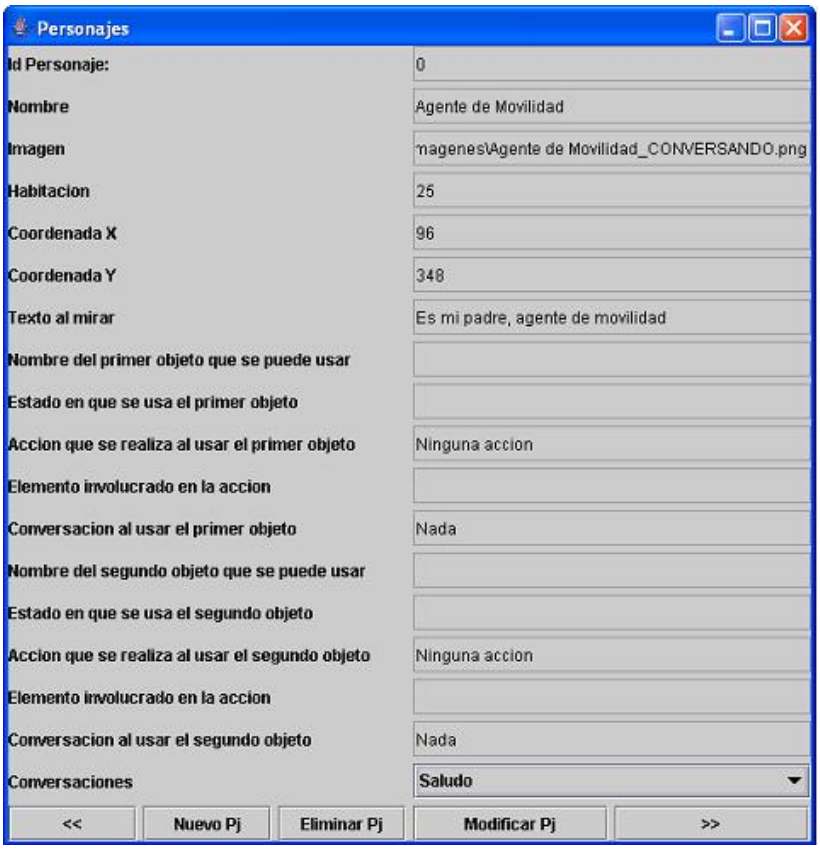
Los ComboBox en los que pone Nada nos permitirán seleccionar una de las conversaciones que hemos creado, sin contar las creadas automáticamente como resultado de una opción de contestación, como conversación que se lanzará al usar un objeto con el personaje. En este caso ahora mismo se podría elegir entre Nada y Saludo pero en el Agente de Movilidad no se usan objetos por lo tanto no se mostrará aquí si no al crear al ProfesorDGT.
El siguiente paso será seleccionar las dos conversaciones nuevas que se crearon y pulsar en Modificar Conversación para rellenarlas, imagen 4.39.



4.39 Rellenando las conversaciones creadas automaticamente

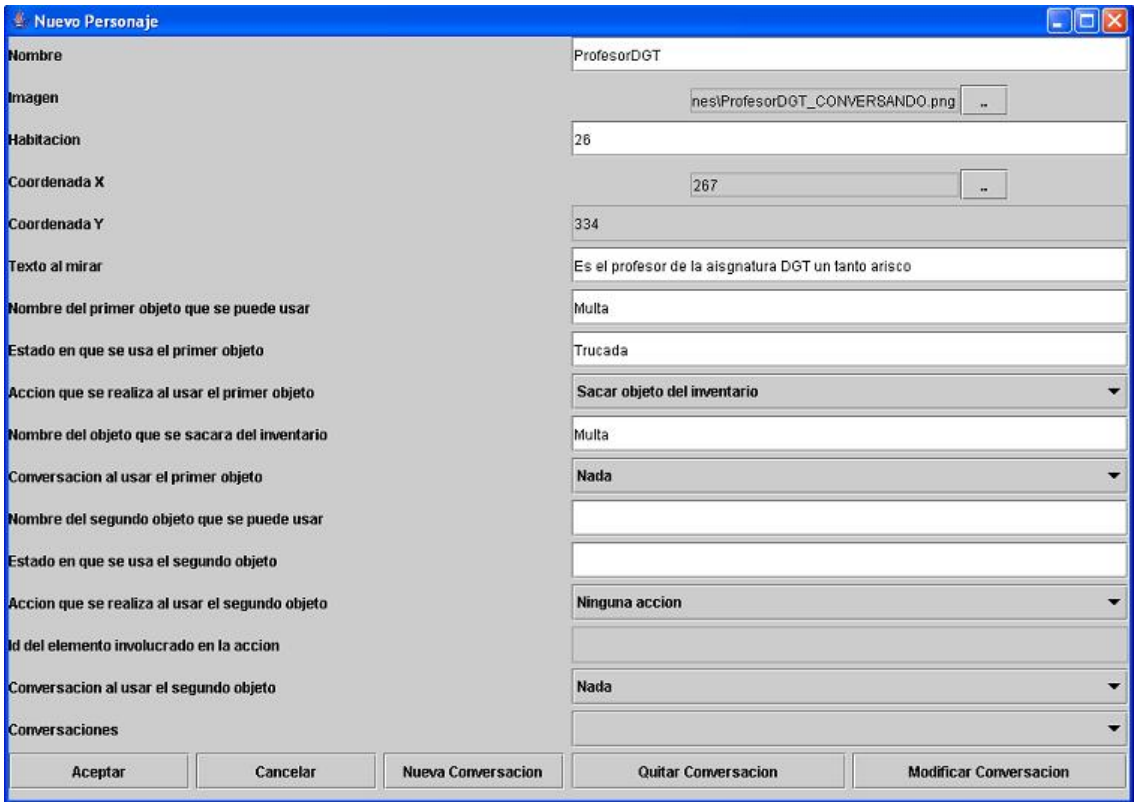
Modificamos la conversación *Pues la verdad es que si*.
Empieza hablando el Agente de Movilidad por lo que indicamos que empieza el Personaje.
Marcamos Si en conversación final porque con esto termina la conversación ya que obtendremos la Multa.
Como acción que se realiza al terminar la conversación elegimos Meter objeto en el inventario y ponemos como id del objeto 0 que es el que se asignó a la multa al crearla.
Por último, añadimos la conversación que se mostrará hasta obtener la multa, aquí podríamos añadir más opciones y hacer esta conversación no final para que el jugador tuviese que convencer al personaje mediante una serie de posibles contestaciones adecuadas, pero lo hacemos directamente para acortar el tutorial.
No mostramos la modificación de *No hasta luego* puesto que es trivial.

Una vez modificada esta conversación pulsamos Aceptar y volveremos al menú de Nuevo Personaje en el que ya tenemos todos los datos necesarios para el correcto funcionamiento del Agente de movilidad. Por lo tanto, pulsamos Aceptar de nuevo y volveremos a la interfaz principal de los personajes, pero esta vez con los datos del Agente de movilidad mostrándose, imagen 4.40.



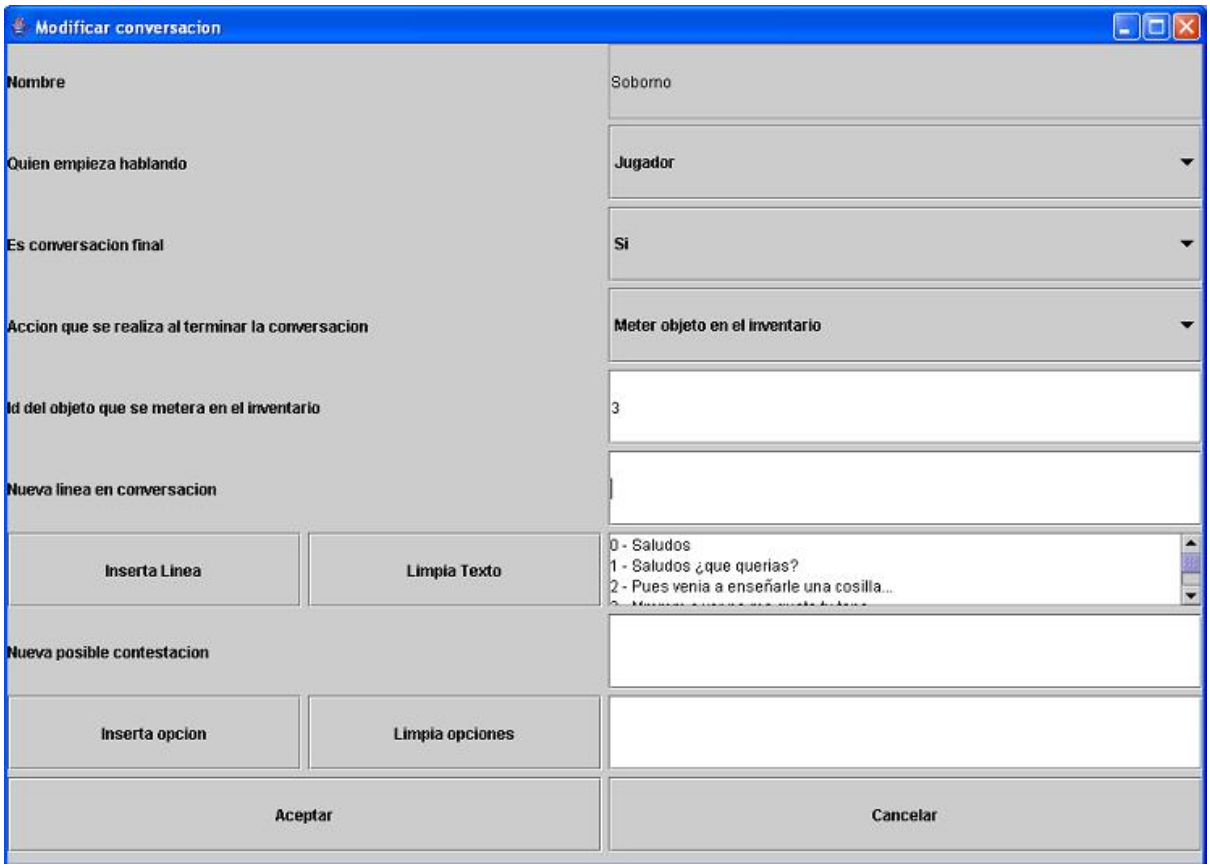
4.40 Datos del Agente de Movilidad recien creado

Lo siguiente será crear el ProfesorDGT, en este caso mostraremos solo los pasos que difieren de la creación del Agente de movilidad en el proceso de creación del personaje.



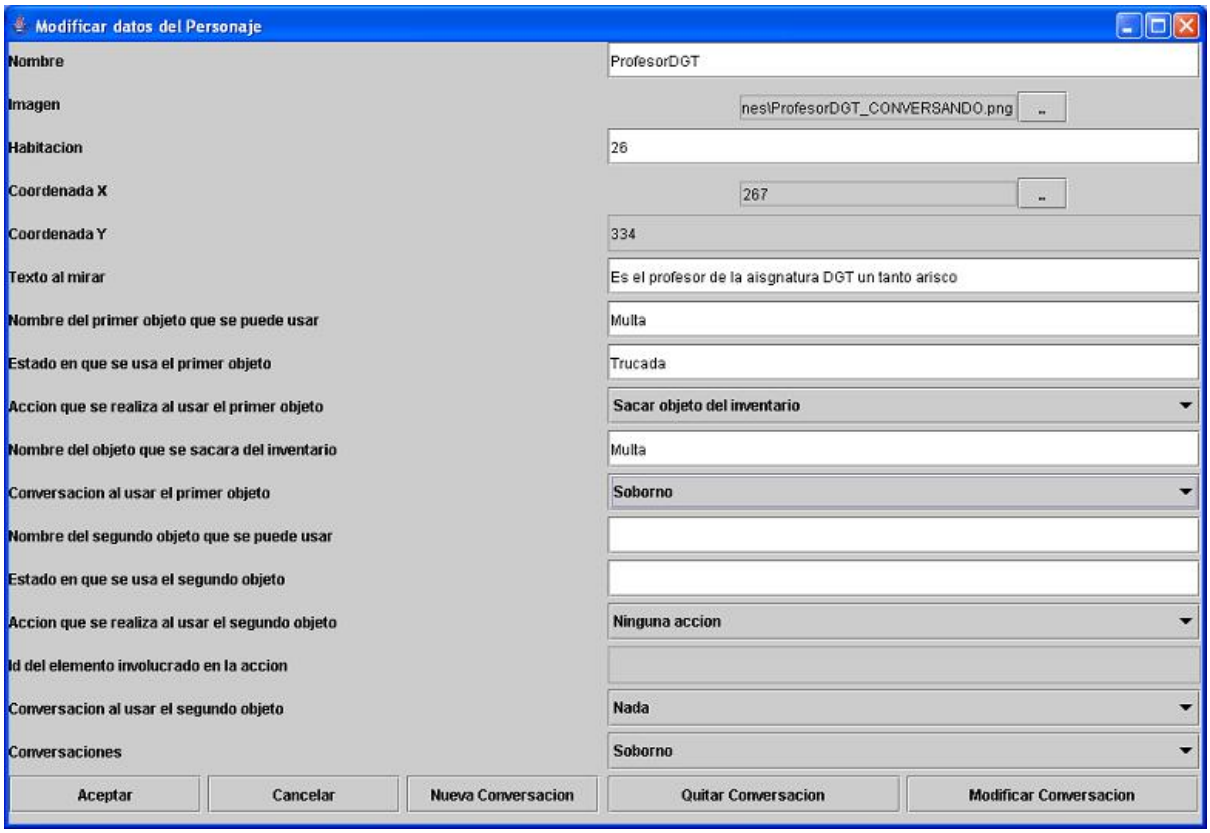
4.41 Datos del ProfesorDGT

En este caso como se aprecia en la imagen 4.41, sí indicamos que se usa un objeto con el personaje, la Multa. Al usarse un objeto hay que indicar en que estado se puede usar y en este caso será en estado Trucada, es decir, tras haberse usado con el bolígrafo para añadir unos cuantos ceros.
Como hemos indicado que se usa un objeto también hay que elegir que acción se realizará tras la conversación que se lance y que conversación será la que se lance. En este caso, la acción será quitar la multa del inventario del personaje puesto que se romperá como motivo del soborno al profesor. La otra acción necesaria para el desarrollo correcto de la misión es meter el aprobado en el inventario, esta acción estará indicada en la conversación que se lanza al usar el objeto y que creamos a continuación, como se muestra en la imagen 4.42.



4.42 Creando la conversación Soborno

Como comentábamos indicamos que se meta un objeto al inventario al terminar la conversación, en este caso el objeto de id 3 que corresponde con el AprobadoDGT, también es conversación final puesto que de no serlo no se permite realizar ninguna acción ya que interrumpiría los eventos de la conversación.
Una vez creada la conversación pulsamos en Aceptar y solo nos quedará asociar esta conversación al uso del objeto que indicabamos anteriormente.

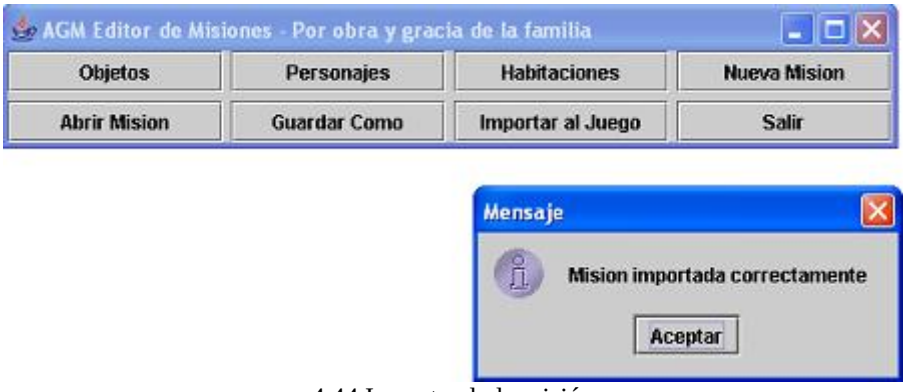


4.43 Asociando la conversación Soborno al uso de la Multa

Como se aprecia en la imagen 4.43 se ha seleccionado la conversación Soborno como la que lanzará al usar el objeto Multa sobre el personaje. Por lo tanto, pulsamos en Aceptar y tendremos el personaje ProfesorDGT terminado.

4.5.6. Importando la misión

Con esto tenemos ya todos los elementos necesarios para el correcto funcionamiento de la misión creados, por lo tanto, solo nos falta importar la misión a AGM. Para ello, volvemos a la interfaz principal del editor y pulsamos en Importar Misión, imagen 4.44, y con ello finaliza el tutorial para crear una misión:



4.44 Importando la misión

4.6. Problemas encontrados durante el desarrollo

Los principales problemas a la hora de desarrollar el editor de misiones surgieron en la conexión con AGM. Varios de los problemas más importantes han sido arreglados por Juan Carlos, tales como las conversaciones entre personajes y jugador y errores en el cambio de habitaciones, sin embargo al estar trabajando a la vez no se disponía de estos arreglos mientras se desarrollaba el editor, lo cual supuso el tener que realizar pequeñas modificaciones para salir al paso y poder realizar pruebas con el editor de manera correcta.

Por ejemplo, al insertar personajes no jugadores desde el Editor se observaba que los vectores de las conversaciones eran correctos pero sin embargo no aparecían muchos mensajes. Se probó escribiendo las conversaciones en consola directamente aparte de en la interfaz en el método escribeConversacion() de la clase ClienteJugador y éstas aparecían correctamente en consola pero no en el juego por lo tanto se llegó a la conclusión de que el causante era el método escribirTexto() de la clase InterfazGrafica.

Se encontró una solución un tanto precaria pero efectiva para poder seguir probando las misiones editadas, se eliminó la comparación *if (escritorTexto.isAlive())* de dicho método y con esto se consiguió que las conversaciones apareciesen normalmente completas aunque en ocasiones demasiado rápido.

Otro problema encontrado consiste en que al hablar con personajes no jugadores en casos en los que se tenga que realizar una acción tras la conversación la acción se realiza pero no se muestra si no desconectamos y volvemos a conectar a la aplicación. Solucionamos este error añadiendo el método de refresco de la habitación al usar conversaciones, esto generó dos pequeños problemas que no se han podido solucionar: al realizar una conversación el personaje puede que se mueva ya que estamos dibujando la habitación de nuevo debido al refresco y en las conversaciones en las que se introducen o quitan objetos tras la conversación estos cambios se realizan durante la misma.
Creemos que todos estos problemas relacionados con las conversaciones se solucionarán al juntar las tres partes que componen el proyecto, puesto que se arregló la perdida de mensajes como se indica en el apartado 2.5.2.

Otro tipo de problema bastante común estaba relacionado con el uso de una única clase para los objetos editados, la clase ObjetoEditor, la aplicación AGM original estaba preparada para trabajar con una clase distinta para cada objeto sin embargo esto era inviable desde el punto de vista del editor.
El uso de una clase común para todos los objetos editados supuso problemas principalmente con la persistencia y con la incorporación de objetos al inventario.

El problema de la persistencia se resolvió rápidamente; en la aplicación original se usaba un LinkedList de nombre PJs en cada clase de objeto, de tal forma que si un personaje cogía el objeto añadía su id a dicho LinkedList y posteriormente podría comprobar al intentar coger de nuevo el objeto si la lista de PJs del objeto ya lo contenía, esto es:

```
If ( !PJs.contains(idPj) )
       return new Acciones(){
              Habitacion h = conseguirHabitacion(idHab);
                   if (interpretaCondicion(h, null, idPJ)) {
              PJs.add(idPj);
              …..
```

La clase ObjetoEditor al ser la misma para todos los objetos editados compartía el LinkedList en todos los objetos, de tal forma que al coger un objeto añadía al jugador a la lista y posteriormente no le dejaba coger un objeto editado distinto puesto que el jugador aparecía en la lista. Lo que se hizo fue concatenar el tipo del objeto (que contenía el nombre del objeto, no de la clase) al id del jugador de tal formas que se distinguía por jugador y por objeto:

```
if (seCoge) {
     if (!(PJs.contains(IdPj + this.tipo))) {
                PJs.add(IdPj + this.tipo);
                guardarLista();
           …..
```

El problema de añadir objetos editados al inventario se solucionó modificando el método meterEnInventario() de la clase Acciones. En la aplicación AGM original a este método se le pasaba por parámetro el nombre de la clase del objeto a meter, por ejemplo:

```
meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.Jabon", null,
                                  "He conseguido una pastilla de jabon.");
```

Y dentro del método se creaba el objeto usando el tipo Class y creando una nueva instancia del objeto:

```
  Class c = Class.forName(tipoObjeto);
  ObjetoNoPersonajeClase objeto = (ObjetoNoPersonajeClase) c.newInstance();
```

Este modo de añadir objetos al inventario era incompatible con los objetos del editor puesto que tendrían todos el mismo comportamiento. Lo que se hizo para solucionar el problema fue añadir un parámetro más al método meterEnInventario(), un objeto de tipo ObjetoEditor que contenía los datos del objeto editado que se debería meter en el inventario:

```
meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.ObjetoEditor", objetoCogido,
mensaje);
```

Y dentro del método se añadió un If para distinguir si el objeto a meter era editado o creado a mano con su propia clase:

```
  if (nombre.compareTo("ObjetoEditor") == 0)
 {
              idO = objetoCogido.getNombre() + idPJ;
              objetoCogido.setIdO(idO);
              hab.meterEnInventario(idPJ, objetoCogido, textoAMostrar);
  }
  else{
        idO = nombre + idPJ;
        objeto.setIdO(idO);
        hab.meterEnInventario(idPJ, objeto, textoAMostrar);
  }
```

5. Bibliografía

Java 2 Volumen II, Características Avanzadas
Horstmann Cay S., Cornell Gary
Editorial Pearson Prentice Hall © 2003

J2EE: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/
JBoss: http://jbossj2ee.pdf
Log4J: http://logging.apache.org/log4j/1.2/manual.html
Swing: http://www.programacion.com/java/tutorial/swing
Awt: http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/parte13/cap13-1.html

6. Palabras clave

J2SE
J2EE
EJB
Bean
Eclipse
Lomboz
JBoss
Aventura Gráfica Multi-jugador
Log4J

7. Cesión de derechos

El equipo de este proyecto (Jaime Agudo, Juan Carlos Castromil y Carlos Pinto) autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado

Jaime Agudo Villanueva

Juan Carlos Castromil Campos

Carlos Pinto Camarero

Madrid a 25 de Septiembre de 2007